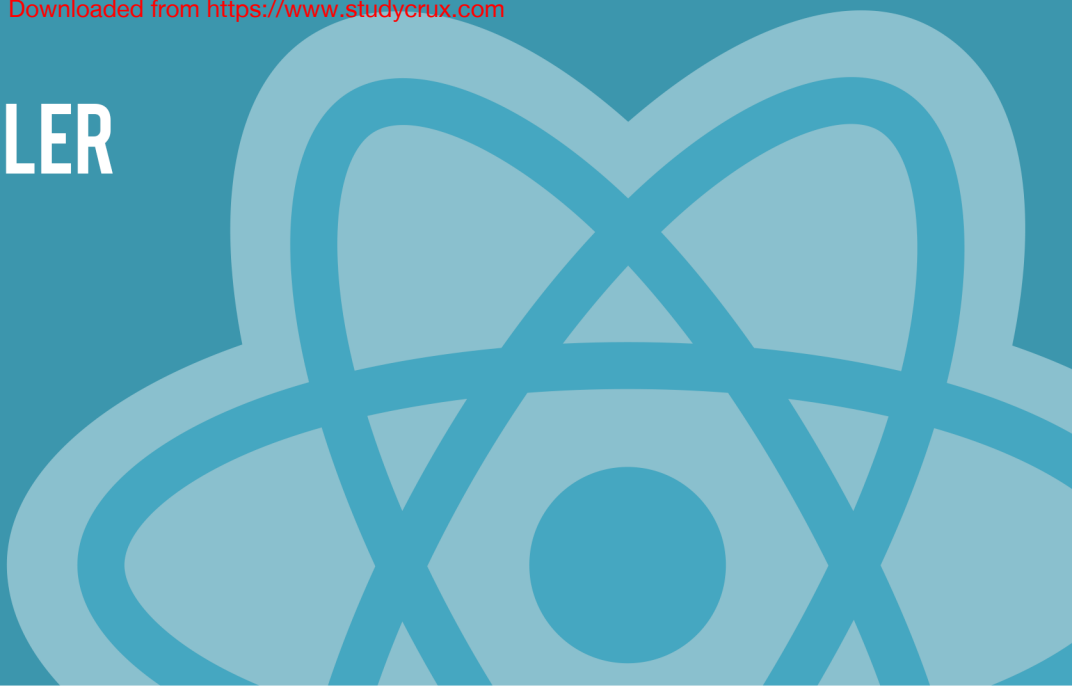
SWIZEC TELLER



REACT + D3v4

Rusable dataviz & animation using modern javascript

The examples in V4 are off the hook.

React+D3v4

Reusable dataviz & games using modern JavaScript

Swizec Teller

This book is for sale at http://leanpub.com/reactd3jses6

This version was published on 2018-01-15

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Tweet This Book!

Please help Swizec Teller by spreading the word about this book on Twitter!

The suggested tweet for this book is:

Gonna build me some cool stuff with #reactd3jsES6

The suggested hashtag for this book is #reactd3jsES6.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

#reactd3jsES6

Also By Swizec Teller

Why programmers work at night

React+d3.js

Contents

Introduction

Hello new friend! :wave:

This is me, Swizec

Welcome to what I hope is the best and most fun React learning resource on the internets. No TODO apps here, just fun interactive stuff that makes you look great at a dinner party.

240 pages, 8 example apps, a bunch of skill building blocks.

By reading this book, you will:

- build a really big interactive data visualization
- build a few silly animations
- learn the basics of D3.js

- master React
- learn the basics of Redux and MobX
- look into rendering rich animations on canvas
- explore React alternatives like Preact and Inferno

You might be thinking *"How the hell does this all fit together to help me build interactive graphs and charts?"*. They're building blocks!

First I will show you the basics. Stuff like how to make React and D3 like each other, how to approach rendering, what to do with state.

Then you will build a big project to see how it all fits together.

When you're comfortable with React and D3 working together, I will show you animation basics. Cool things like how to approach updating components with fine grained control and how to do transitions. I'll show you that tricks from the gaming industry work here too.

Short on time? Don't worry. The basic examples are built on Codepen, a web-based IDE. Read the explanation, run the code in your browser, master enough React to be dangerous.

In about an hour, you'll know React well enough to explore on your own, to sell the tech to your boss and your team, and to assess if the React and D3 combination fits your needs.

I believe learning new tech should be exciting.

Foreword

I wrote the first version of this book in Spring 2015 as a dare. Can I learn a thing and write a book in a month? Yes.

That first version was 91 pages long and sold a few hundred copies. Great success! This version is 240 pages and sold over a hundred copies in preorders alone.

That makes me happy. If you're one of those early supporters, thank you! You're the best.

Here's what I had to say about the first version:

> I wrote this book for you as an experiment. The theory we're testing is that technical content works better as a short e-book than as a long article.
>
> You see, the issue with online articles is that they live and die by their length. As soon as you surpass 700 words, everything falls apart. Readers wander off to a different tab in search of memes and funny tweets, never to come back.
>
> This hurts my precious writer's ego, but more importantly, it sucks for you. If you're not reading, you're not learning.
>
> So here we both are.
>
> I'm counting on you to take some time away from the internet, to sit down and read. I'm counting on you to follow the examples. I'm counting on you to *learn*.
>
> You're counting on me to have invested more time, effort, and care in crafting this than I would have invested in an article. I have.
>
> I've tried to keep this book as concise as possible. iA Writer estimates it will take you about an hour to read *React+d3.js*, but playing with the examples might take some time, too.

All of that still rings true. People are distracted, the internet is shiny, and it's hard to keep readers for more than 5 minutes at a time.

I don't know how fast you read, but I don't *really* learn things with a 5-minute skim. I need to sit down, try stuff out, play around.

That's what I'm hoping you'll do with this book. Sit down, play with code, try the examples, and have some fun.

If you're short on time: Read the theory. You'll get the gist and learn the basics in about an hour. I promise :)

Why you should read React + D3v4

After an hour with *React + D3v4*, you'll know how to make React and D3.js play together. You'll know how to create composeable data visualizations. You're going to understand *why* that's a good idea, and you will have the tools to build your own library of interactive visualization components.

It's going to be fun! I believe learning new tech should be exciting, so I did my best to inject joy whenever I could.

What you need to know

I'm going to assume you already know how to code and that you're great with JavaScript. Many books have been written to teach the basics of JavaScript; this is not one of them.

If you're struggling with modern JavaScript syntax, I suggest visiting my Interactive ES6 Cheatsheet[1]

I'm also going to assume some knowledge of D3.js. Since it isn't a widely-used library, I'm going to explain the specific bits that we use. If you want to learn D3.js in depth, you should the third edition of my first ever book, D3.js 4.x Data Visualization - Third Edition[2]. Written by Ændrew Rininsland with some bits left from yours truly.

React is a new kid on the block... well it's been 2 years but still. You'll be fine even if this is your first time looking at React. This book is more about learning React than it is about learning D3.

All of the examples in React + D3v4 are written in modern JavaScript. Most code is ES6 – ECMAScript2015 – some might be 2016, maybe even 2017. I find it hard to keep track to be honest.

I'm going to explain any new syntax that we use.

How to read this book

Relax. Breathe. You're here to learn. I'm here to teach. I promise Twitter and Facebook will still be there when we're done.

For now, it's just you and some fun code. To get the most out of this material, I suggest two things:

1. Try the example code yourself. Don't just copy-paste; type it and execute it. Execute it frequently. If something doesn't fit together, look at the linked Github repositories here[3]. Each example has one.
2. If you already know something, skip that section. You're awesome. Thanks for making this easier.

React + D3v4 is based on code samples. They look like this:

[1] https://es6cheatsheet.com

[2] https://www.packtpub.com/web-development/d3js-4x-data-visualization-third-edition

[3] https://github.com/Swizec/h1b-software-salaries

Some code

```
let foo = 'bar';
foo += 'more stuff';
```

Each code sample starts with a commented out file path. That's the file you're editing. Like this:

Code samples have file paths

```
// ./src/App.js

class App ...
```

Commands that you should run in the terminal start with an `$` symbol, like this:

```
$ npm start
```

Why React and D3.js

React is Facebook's and Instagram's approach to writing modern JavaScript front-ends. It encourages building an app out of small, re-usable components. Each component is self-contained and only knows how to render a small bit of the interface.

The catch is that many frameworks have attempted this: everything from Angular to Backbone and jQuery plugins. But where jQuery plugins quickly become messy, Angular depends too much on HTML structure, and Backbone needs a lot of boilerplate, React has found a sweet spot.

I have found it a joy to use. Using React was the first time I have ever been able to move a piece of HTML without having to change any JavaScript.

D3.js is Mike Bostock's infamous data visualization library. It's used by The New York Times along with many other sites. It is the workhorse of data visualization on the web, and many charting libraries out there are based on it.

But D3.js is a fairly low-level library. You can't just say *"I have data; give me a bar chart"*. Well, you can, but it takes a few more lines of code than that. Once you get used to it though, D3.js is a joy to use.

Just like React, D3.js is declarative. You tell it *what* you want instead of *how* you want it. It gives you access straight to the SVG so you can manipulate your lines and rectangles at will. The issue is that D3.js isn't that great if all you want are charts.

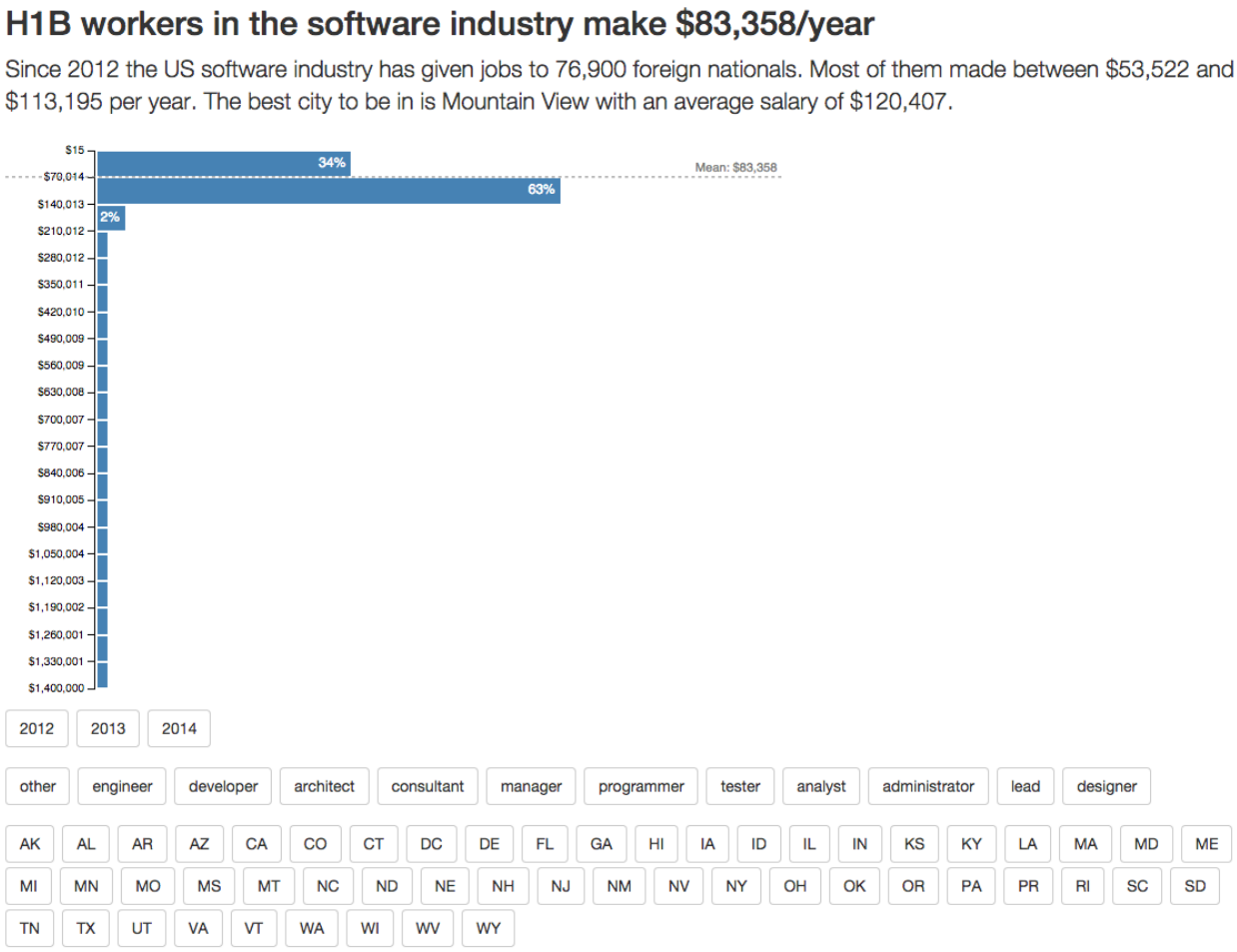
This is where React comes in. For instance, once you've created a histogram component, you can always get a histogram with `<Histogram {...params} />`.

Doesn't that sound like the best? I think it's amazing.

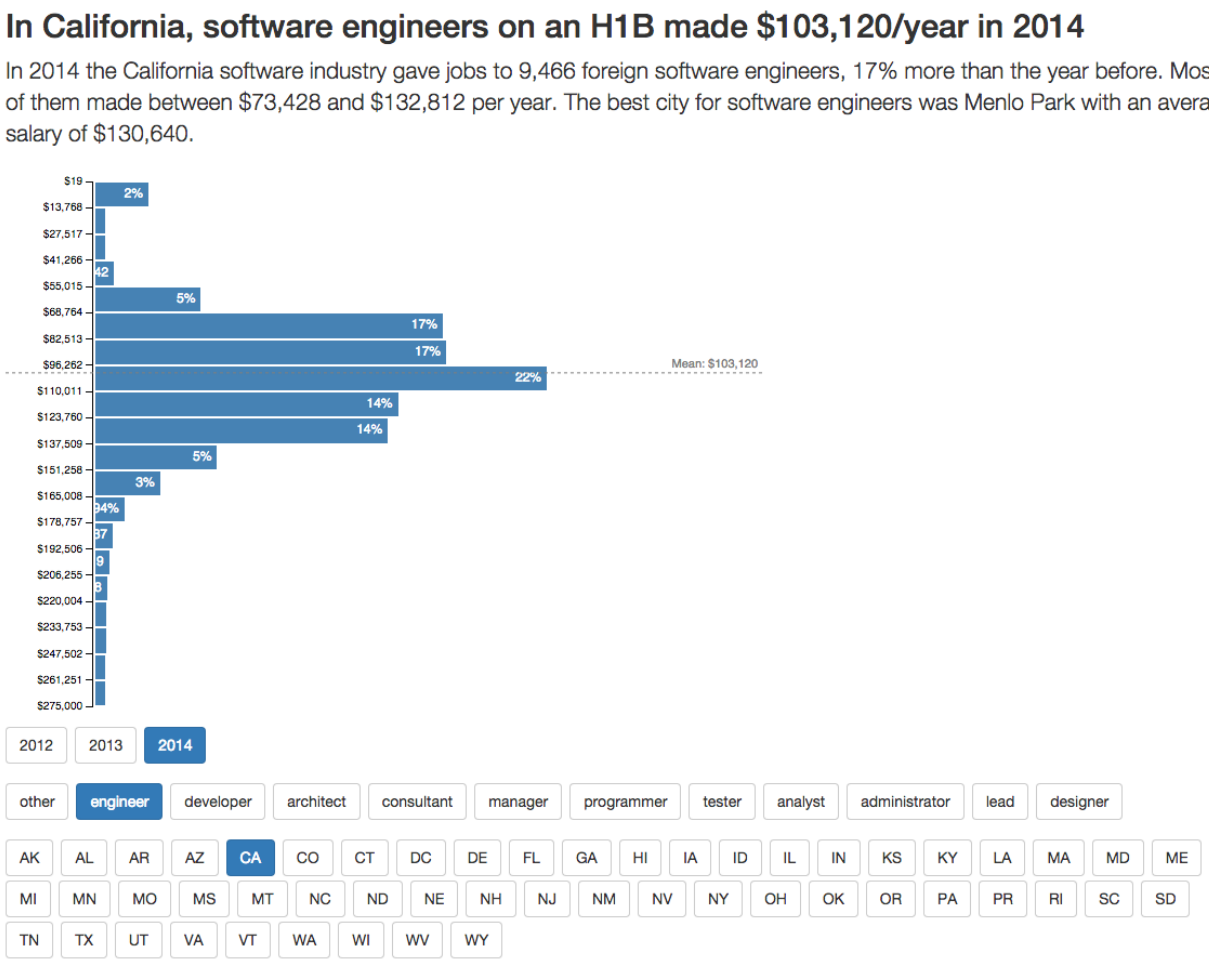
It gets even better. With React, you can make various graph and chart components build off of the same data. This means that when your data changes, the whole visualization reacts.

Your graph changes. The title changes. The description changes. Everything changes. Mind = blown.

Look how this H1B salary visualization changes when the user picks a subset of the data to look at.



Default H1B histogram



Changes after user picks a data subset

React + d3.js: a powerful combination indeed.

Buzzword soup explained

We're going to use a lot of buzzwords in this book. Hell, we've used some already. Most will get a thorough explanation further on, but let me give you a quick rundown.

- **Babel**, a JavaScript transpiler that makes your code work on all browsers.
- **ECMAScript2016**, official name for the part of ES7 we're getting in 2016
- **ES5**, any JavaScript features that existed before June 2015
- **ES6/ECMAScript2015**, any JavaScript features released as part of the new standard in June 2015 (think `=>` and stuff)
- **ES7**, the standard we got in 2016, but also a catch-all for future JavaScript features
- **ES6+**, an umbrella term for "modern JavaScript"; anything new since 2015
- **fat arrows**, a new way to define functions in ES6 (`=>`)
- **Git**, a version control system. It's pretty popular, you probably know it :)
- **H1B**, a popular type of work visa in the United States
- **JSX**, a language/syntax that lets you use `<a href="/example">` as a native part of JavaScript
- **Mustache**, a popular way to write HTML templates for JavaScript code. Uses `{{ ... }}` syntax.
- **npm**, most popular package manager for JavaScript libraries
- **props**, component properties set when rendering
- **state**, a local dictionary of values available in most components
- **functional stateless components**, React components expressed as pure functions that take props and return markup
- **Webpack**, a module packager for JavaScript. Makes it more convenient to organize code into multiple files. Provides cool plugins.

JSX

We're going to write our components in JSX, a JavaScript syntax extension that lets us treat XML-like data as normal code. You can use React without JSX, but I think JSX makes React's full power easier to use.

The gist of JSX is that we can use any XML-like string just like it was part of JavaScript. No Mustache or messy string concatenation necessary. Your functions can return HTML, SVG, or XML.

For instance, code that renders one of our first examples – a color swatch – looks like this:

Render a color swatch

```
ReactDOM.render(
        <Colors width="400" />,
        document.getElementById('svg')
);
```

Which compiles to:

JSX compiled result

```
ReactDOM.render(
        React.createElement(Colors, {width: "400"}),
        document.querySelectorAll('svg')[0]
);
```

As you can see, HTML code translates to `React.createElement` calls with attributes translated into a property dictionary. The beauty of this approach is two-pronged: you can use React components as if they were HTML tags, and HTML attributes can be anything.

You'll see that anything from a simple value to a function or an object works equally well.

I used to be a fan of the One Language Per File principle, but using JSX has made my code much easier to work with. Especially in a small team where I do everything - the code, the markup, the styles - having the relevant code colocated helps a lot.

But I wouldn't let a designer edit my HTML in JavaScript, for instance. Can't see that ending well.

Visualizing data with React and d3.js

Welcome to the main part of React + D3v4. We're going to talk a little theory, learn some principles, and then get our hands dirty with a few examples. Through this book you're going to build:

- A few small components in Codepen
- A choropleth map
- An interactive histogram
- A rainbow snake
- An animated alphabet
- A simple particle generator with Redux
- A particle generator pushed to 20,000 elements with canvas
- Billiards simulation with MobX and canvas
- A dancing fractal tree

Looks random, right? Bear with me.

Examples build on each other in complexity. **The first** teaches you how to make a static data visualization component and shows you an approach to declarative data visualization. **The second** adds interactivity and components interacting with each other in order to teach you about a simple approach to state management. **The third** shows you how to use transitions to build simple animations. **The fifth** shows you complex transitions with entering and exiting components. **The fifth** shows you how to do complex animation using a game loop principle. **The sixth**, **seventh**, **and eighth** show you how to approach speed optimization to smoothly animate thousands of elements.

Throughout our examples, we're going to use **React 15**, compatible with **React 16 Fiber**, **D3v4**, and **ES6+**. In the particle generator, we're also going to use **Redux** to drive the game loop and **Konva** for canvas manipulation. The billiards simulation uses **MobX** so you can compare it to Redux and learn both.

Don't worry if you're not comfortable with ES6 syntax yet. By the end of this book, you're gonna love it!

Until then, here's an interactive cheatsheet: es6cheatsheet.com[4]. It uses code samples to compare the ES5 way with the ES6 way so you can brush up quickly.

In the interest of looking towards the future, we'll check out **Preact** and **Inferno** in the final fractal example. They're new React-like libraries that promise better performance and smaller file sizes.

[4] https://es6cheatsheet.com/

Their creators – Jason Miller and Dominic Gannaway - graciously forked my fractal tree example and built the same thing in their respective libraries. Thank you!

—-

Before we begin our examples, let's talk about how React and D3 fit together and how we're going to structure our apps. If you prefer to get your hands dirty first and ask questions later, skip this section and jump to the examples.

This section is split into five chapters:

- Basic Approach
- Blackbox Components
- Full Feature Integration
- State Handling Architecture
- Structuring your React App

The basic approach

Our visualizations are going to use SVG - an XML-based image format that lets us describe images in terms of mathematical shapes. For example, the source code of an 800x600 SVG image with a rectangle looks like this:

SVG rectangle

```
<svg width="800" height="600">
    <rect width="100" height="200" x="50" y="20" />
</svg>
```

These four lines create an SVG image with a black rectangle at coordinates `(50, 20)` that is 100x200 pixels large. Black fill with no borders is default for all SVG shapes.

SVG is perfect for data visualization on the web because it works in all browsers, renders without blurring or artifacts on all screens, and supports animation and user interaction. You can see examples of interaction and animation later in this book.

But SVG can get slow when you have many thousands of elements on screen. We're going to solve that problem by rendering bitmap images with canvas. More on that later.

—-

Another nice feature of SVG is that it's just a dialect of XML - nested elements describe structure, attributes describe the details. They're the same principles that HTML uses.

That makes React's rendering engine particularly suited for SVG. Our 100x200 rectangle from before looks like this as a React component:

A simple rectangle in React

```
const Rectangle = () => (
    <rect width="100" height="200" x="50" y="20" />
);
```

To use this rectangle component in a picture, you'd use a component like this:

Rect component in a picture

```
const Picture = () => (
    <svg width="800" height="600">
        <Rectangle />
    </svg>
);
```

You're right. This looks like tons of work for a static rectangle. But look closely. Even if you know nothing about React and JSX, you can look at that code and see that it's a `Picture` of a `Rectangle`.

Compare that to a pure D3 approach:

A static rectangle in d3.js

```
d3.select("svg")
  .attr("width", 800)
  .attr("height", 600)
  .append("rect")
  .attr("width", 100)
  .attr("height", 200)
  .attr("x", 50)
  .attr("y", 20);
```

It's elegant, it's declarative, and it looks like function call soup. It doesn't scream *"Rectangle in an SVG"* to me as much as the React example does.

You have to take your time and read the code carefully: first, we `select` the `svg` element, then we add attributes for `width` and `height`. After that, we `append` a `rect` element and set its attributes for `width`, `height`, `x`, and `y`.

Those 8 lines of code create HTML that looks like this:

HTML of a rectangle

```
<svg width="800" height="600">
    <rect width="100" height="200" x="50" y="20" />
</svg>
```

Would've been easier to just write the HTML, right? Yes, for static images, you're better off using Photoshop or something then exporting to SVG.

Either way, dealing with the DOM is not D3's strong suit. There's a lot of typing, code that's hard to read, it's slow when you have thousands of elements, and it's often hard to keep track of which

elements you're changing. D3's enter-update-exit cycle is great in theory, but I personally never found it easy to use.

If you don't completely understand what I just said, don't worry. We'll cover the enter-update-exit cycle in the animations example. Don't worry about D3 either. **I know it's hard**. I've written two books about it, and I still spend as much time reading the docs as writing the code. There's much to learn, and I'll explain everything as we go along.

D3's strong suit is its ability to do everything other than the DOM. There are many statistical functions, great support for data manipulation, and a bunch of built-in data visualizations. **D3 can calculate anything for us**. **All we have to do is draw it out**.

That's why we're going to follow this approach:

- React owns the DOM
- D3 calculates properties

This way, we can leverage React for SVG structure and rendering optimizations and D3 for all its mathematical and visualization functions.

Now let's look at two different ways to put them together: blackbox and full-feature.

Blackbox Components

Blackbox components are the simplest way to integrate D3 and React. You can think of them as wrappers around D3 visualizations.

You can take any D3 example from the internets or your brain, wrap it in a thin React component, and it Just Works™. Yes, we go against what I just said and let D3 control a small part of the DOM tree.

We throw away most of React's power, but we gain a quick way to get things working.

I call the approach *"blackbox"* because React's engine can't see inside your component, can't help with rendering, and has no idea what's going on. From this point onward in the DOM tree, you are on your own. It sounds scary, but it's okay if you're careful.

Here's how it works:
- React renders an anchor element
- D3 hijacks it and puts stuff in

You have to manually re-render on props and state changes. You're also throwing away and recreating your component's entire DOM on each re-render.

Manual re-rendering is not as annoying as it sounds, but the inefficiency can get pretty bad with complex visualizations. Use this technique sparingly.

A quick blackbox example - a D3 axis

Let's build an axis component. Axes are the perfect use-case for blackbox components. D3 comes with an axis generator bundled inside, and they're difficult to build from scratch.

They don't *look* difficult, but there are many tiny details you have to get *just right*.

D3's axis generator takes a scale and some configuration to render an axis for us. The code looks like this:

A pure D3 axis

```
const scale = d3.scaleLinear()
                .domain([0, 10])
                .range([0, 200]);
const axis = d3.axisBottom(scale);

d3.select('svg')
  .append('g')
  .attr('transform', 'translate(10, 30)')
  .call(axis);
```

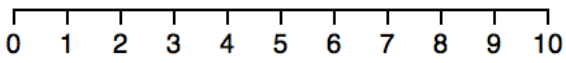
If this code doesn't make any sense, don't worry. There's a bunch of D3 to learn, and I'll help you out. If it's obvious, you're a pro! This book will be much quicker to read.

We start with a linear scale that has a domain `[0, 10]` and a range `[0, 200]`. You can think of scales as mathematical functions that map a domain to a range. In this case, calling `scale(0)` returns `0`, `scale(5)` returns `100`, `scale(10)` returns `200`. Just like middle school mathematics.

We create an axis generator with `axisBottom`, which takes a `scale` and is going to generate a `bottom` oriented axis – numbers below the line. You can also tweak settings for the number of ticks, their sizing, and their spacing.

Equipped with an `axis` generator, we `select` the `svg` element, append a grouping element, use a `transform` attribute to move it `10`px to the right and `30`px down, and invoke the generator with `.call()`.

It creates a small axis:



Simple axis

You can play around with this example on CodePen here[5]. Try changing the scale type.

A quick blackbox example - a React+D3 axis

Now let's say we want to use that same axis code but as a React component. The simplest way is to use a blackbox component approach like this:

[5] https://codepen.io/swizec/pen/YGoYBM

React blackbox axis

```
class Axis extends Component {
        componentDidMount() { this.renderAxis() }
        componentDidUpdate() { this.renderAxis() }

        renderAxis() {
                const scale = d3.scaleLinear()
                          .domain([0, 10])
                   .range([0, 200]);
            const axis = d3.axisBottom(scale);

                d3.select(this.refs.g)
                  .call(axis);
        }

        render() {
            return <g transform="translate(10, 30)" ref="g" />
        }
}
```

Oh man! So much code! Is this really worth it? Yes, for the other benefits of using React in your dataviz. You'll see :)

We created an `Axis` component that extends React base `Component` class. We can't use functional stateless components because we need lifecycle hooks. More on those later.

Our component has a `render` method, which returns a grouping element (`g`) moved 10px to the right and 30px down using the `transform` attribute. Same as before.

We added a `ref` attribute, which lets us reference elements in our component via `this.refs`. This makes D3 integration cleaner, and it probably works with React-native. I haven't tried yet.

The body of `renderAxis` should look familiar. It's where we put code from the pure D3 example. Scale, axis, select, call. There's no need to append a grouping element; we're already there with `this.refs.g`.

For the manual re-rendering part, we call `renderAxis` in `componentDidUpdate` and `componentDidMount`. This ensures that our axis re-renders every time React's engine decides to render our component. On state and prop changes usually.

That wasn't so bad, was it? You can try it out on CodePen here[6].

To make our axis more useful, we could get the scale and axis orientation from props. We'll do that for scales in our bigger project.

[6] https://codepen.io/swizec/pen/qazxPz

A D3 blackbox higher order component – HOC

After the blackbox axis example above, you'd be right to think something like *"Dude, that looks like it's gonna get hella repetitive. Do I really have to do all that every time?"*

Yes, you do. But! We can make it easier with a higher order component - a HOC.

Higher order components are one of the best ways to improve your React code. When you see more than a few components sharing similar code, it's time for a HOC. In our case, that shared code would be:

- rendering an anchor element
- calling D3's render on updates

With a HOC, we can abstract that away into something called a class factory[7]. It's an old concept coming back in vogue now that JavaScript has classes.

You can think of it as a function that takes some params and creates a class – a React component. Another way to think about HOCs is that they're React components wrapping other React components and a function that makes it easy.

Let's build a HOC for D3 blackbox integration. We'll use it in the main example project.

A `D3blackbox` HOC looks like this:

React blackbox HOC

```
function D3blackbox(D3render) {
        return class Blackbox extends React.Component {
                componentDidMount() { D3render.call(this); }
                componentDidUpdate() { D3render.call(this) }

                render() {
            const { x, y } = this.props;
            return <g transform={`translate(${x}, ${y})`} ref="anchor" />;
                }
        }
}
```

[7] https://en.wikipedia.org/wiki/Factory_method_pattern

You'll recognize most of that code from earlier. We have a `componentDidMount` and`componentDidUpdate` lifecycle hooks that call `D3render` on component updates. This used to be called `renderAxis`. `render` renders a grouping element as an anchor into which D3 can put its stuff.

Because `D3render` is no longer a part of the component, we have to use `.call` to give it the scope we want: this class, or rather `this` instance of the `Backbone` class.

We've also made some changes to make `render` more flexible. Instead of hardcoding the `translate()` transformation, we take `x` and `y` props. `{ x, y } = this.props` takes `x` and `y` out of `this.props` using object decomposition, and we used ES6 string templates for the `transform` attribute.

Consult the ES6 cheatsheet[8] for details on that.

Using our new `D3blackbox` HOC to make an axis looks like this:

React blackbox HOC

```
1  const Axis = D3blackbox(function () {
2      const scale = d3.scaleLinear()
3                      .domain([0, 10])
4                      .range([0, 200]);
5      const axis = d3.axisBottom(scale);
6
7      d3.select(this.refs.anchor)
8        .call(axis);
9  });
```

It's the same code as we had in `renderAxis` before. The only difference is that the function is wrapped in a `D3blackbox` call. This turns it into a React component.

I'm not 100% whether wrapping a function in a React component counts as a real HOC, but let's roll with it. More proper HOCs are React components wrapped in components.

You can play with this example on Codepen here[9].

[8] https://es6cheatsheet.com/

[9] https://codepen.io/swizec/pen/woNjVw

Full-feature Integration

As useful as blackbox components are, we need something better if we want to leverage React's rendering engine. We're going to look at full-feature integration where React does the rendering and D3 calculates the props.

To do that, we're going to follow a 3-part pattern:
- set up D3 objects as class properties
- update D3 objects when component updates
- output SVG in `render()`

It's easiest to show you with an example.

Let's build a rectangle that changes color based on prop values. We'll render a few of them to make a color scale.

Yes, it looks like a trivial example, but color-as-information is an important concept in data visualization. We're going to use it later to build a choropleth map of household income in the US.

I suggest following along in Codepen for now. Here's one I set up for you[10]. It contains the final solution, so you can follow along and nod your head. I'll explain each part.

A color scale

We start with a `Swatch` component that draws a rectangle and fills it with a color.

Swatch component

```
const Swatch = ({ color, width, x }) => (
        <rect width={width}
                        height="20"
                        x={x}
                        y="0"
                        style={{fill: color}} />
);
```

Looks like our earlier components, doesn't it? It's exactly the same: a functional stateless component that draws a `rect` element with some attributes - dimensions, position, and `fill` style.

[10] https://codepen.io/swizec/pen/oYNvpQ

Note that `style` is a dictionary, so we specify it with double curly braces: outer braces for a dynamic value, inner braces for a dictionary.

Then we need a `Colors` component. It follows the full-featured integration structure: D3 objects as properties, an `updateD3` function, plus some wiring for updates and rendering.

Colors component, pt1

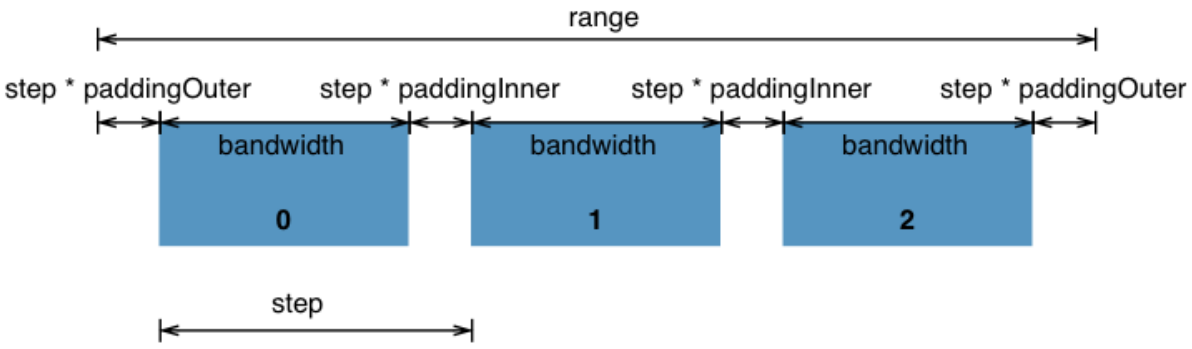
```
1 class Colors extends Component {
2     colors = d3.schemeCategory20;
3     width = d3.scaleBand()
4                                     .domain(d3.range(20));
```

We start by inheriting from `Component` and defining defaults for D3 objects. `this.colors` is one of D3's predefined color scales[11]. `schemeCategory20` is a scale of 20 colors designed for categorization. It seemed like a good example, and you're welcome to try others.

`this.width` is a D3 scale designed for producing bands, `d3.scaleBand`. As mentioned earlier, scales map domains to ranges. We know our domain is 20 colors, so we can statically set the domain as `[1, 2, 3, ..., 20]` with `d3.range(20)`.

`d3.range` generates a counting array, by the way. We'll use that often.

We'll use `this.width` to calculate widths and positions of our color swatches. Here's a picture from D3 docs to help you visualize what `scaleBand` does:



Band Scale from D3 docs

Unlike the domain, our range is dynamic so that we can use props to define the width of our color scale. This makes the component more reusable.

[11] https://github.com/d3/d3-scale/blob/master/README.md#schemeCategory10

Colors component, pt2

```
componentWillMount() {
    this.updateD3(this.props);
}

componentWillUpdate(newProps) {
    this.updateD3(newProps);
}

updateD3(props) {
    this.width.range([0, props.width]);
}
```

`componentWillMount` and `componentWillUpdate` are component lifecycle hooks. Can you guess when they run?

`componentWillMount` runs just before React's engine inserts our component into the DOM, and `componentWillUpdate` runs just before React updates it. That happens on any prop change or `setState` call.

Both of them call our `updateD3` method with the new value of props. We use it to update `this.width` scale's range. Doing so keeps the internal state of D3 objects in sync with React's reality. Without it, our component might render stale data.

Finally, we render a set of color swatches.

Colors component, pt3

```
render() {
        return (
                <g>
            {d3.range(20).map(i => (
                <Swatch color={this.colors[i]}
                        width={this.width.step()}
                        x={this.width(i)} />
             ))}
                </g>
        )
}
```

We create a grouping element to fulfill React's one child per component requirement, then render 20 swatches in a loop. Each gets a `color` from `this.colors` and a `width` and `x` from `this.width`.

After inserting into the DOM with `ReactDOM`, we get a series of 20 colorful rectangles.

20 color swatches

Try changing the `width="400"` property of `<Colors />`. You'll see D3's `scaleBand` and our update wiring ensure the color strip renders correctly. For more fun, try changing the `Colors` component so it takes the color scale as a prop, then rendering multiple instances of `<Colors />` side-by-side.

Here's the playground again: CodePen[12]

As an exercise, try to add another row of swatches, but rendered in reverse.

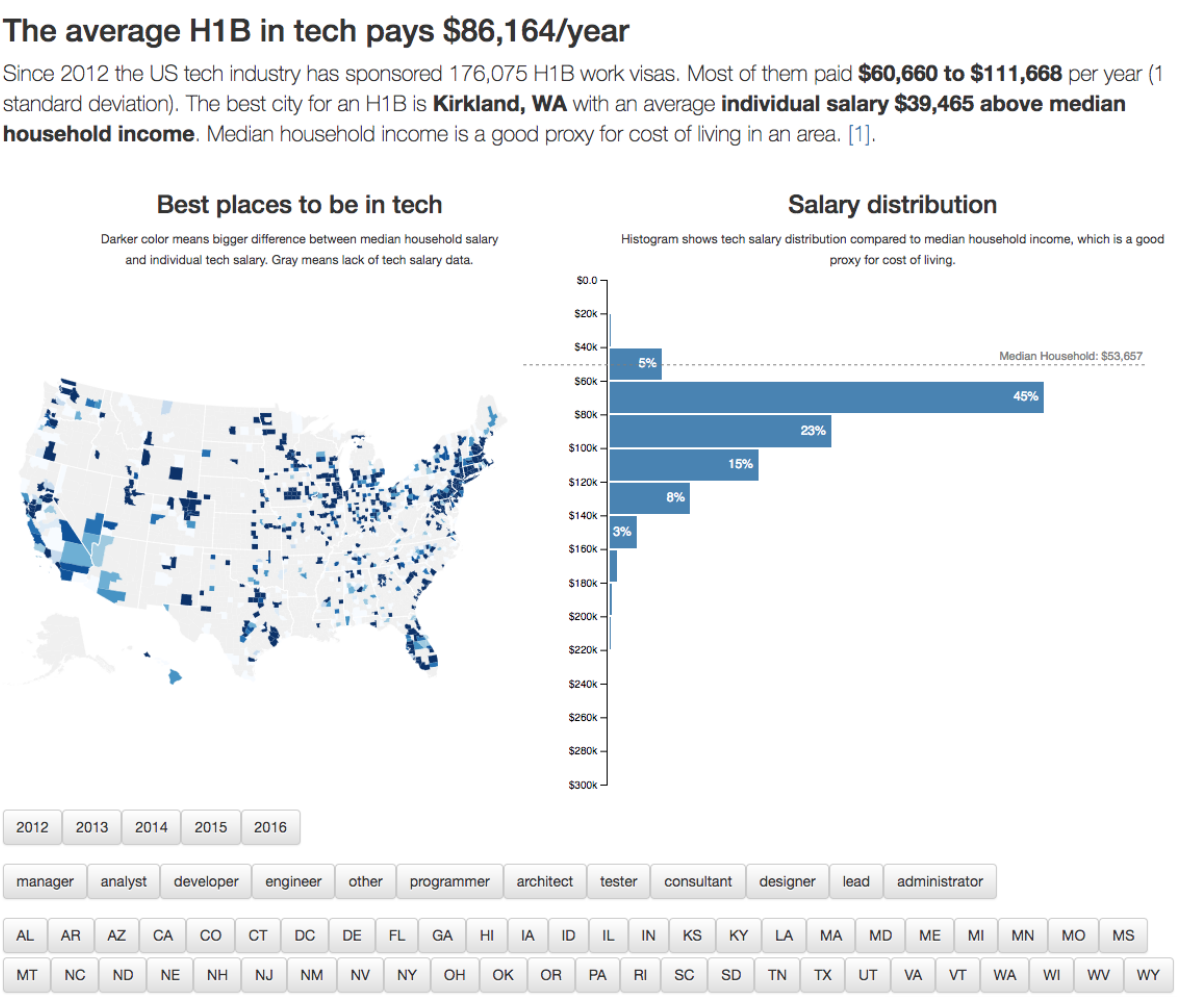
[12] https://codepen.io/swizec/pen/oYNvpQ

You're awesome

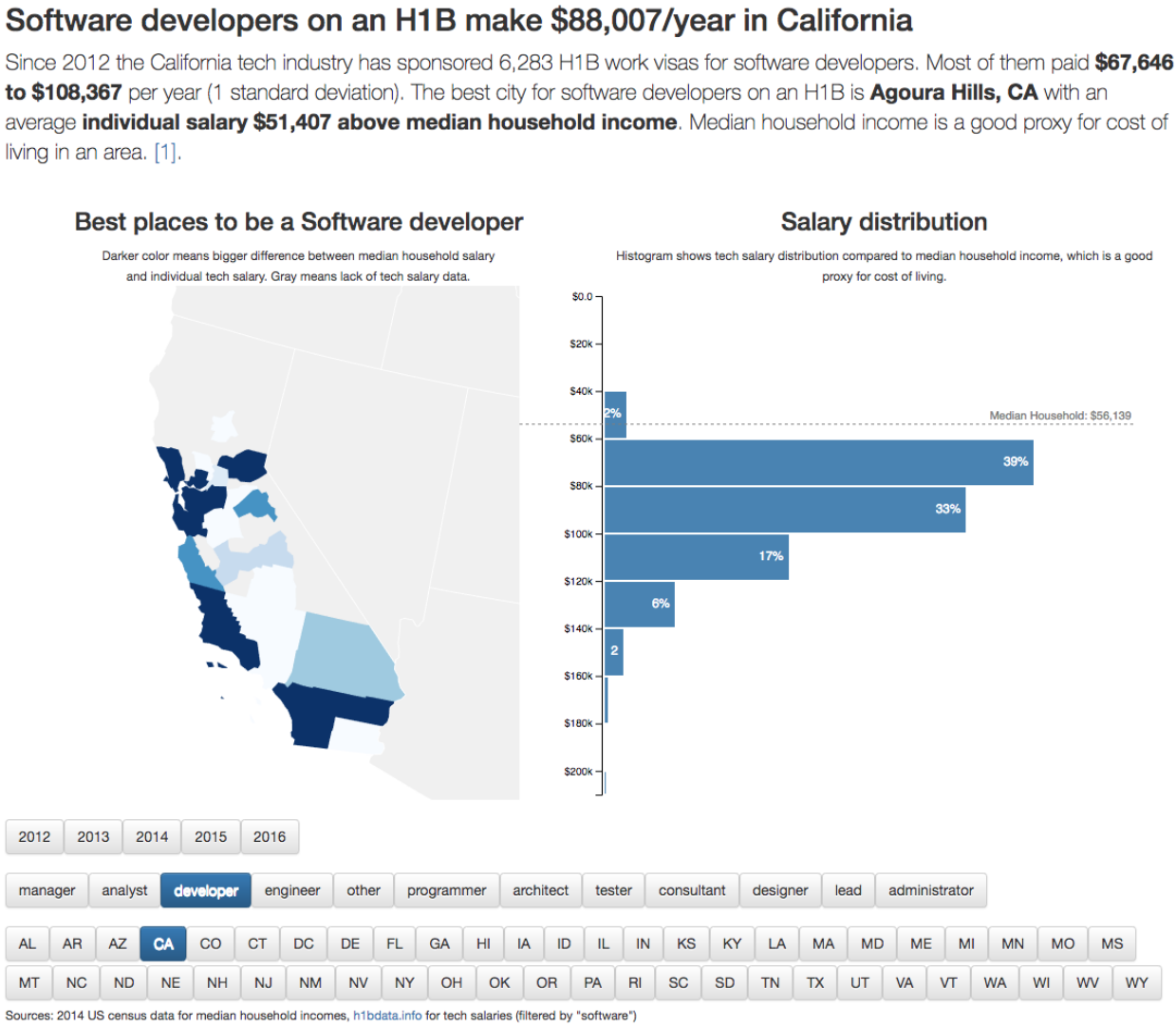
You know the basics! You can take any D3 example from the internets and wrap it in a React component, *and* you know how to build React+D3 components from scratch. You're amazing. High five! :raised_hand_with_fingers_splayed:

The rest of this book is about using these concepts and pushing them to the limits of practicality. We're going to build an interactive visualization of tech salaries compared to median household income. Why? Because it piqued my interest, and because it shows why you should call yourself an engineer, not a programmer or a developer. **You're an engineer**. Remember that.

Throughout the example, you'll learn more details of D3, tidbits from React, and the animation chapter is going to blow your mind. It's gonna be fun!



Default view



After a click

State Handling Architecture

Before I can set you loose on the world, we should talk about managing state. It's where most engineers shoot themselves in the foot.

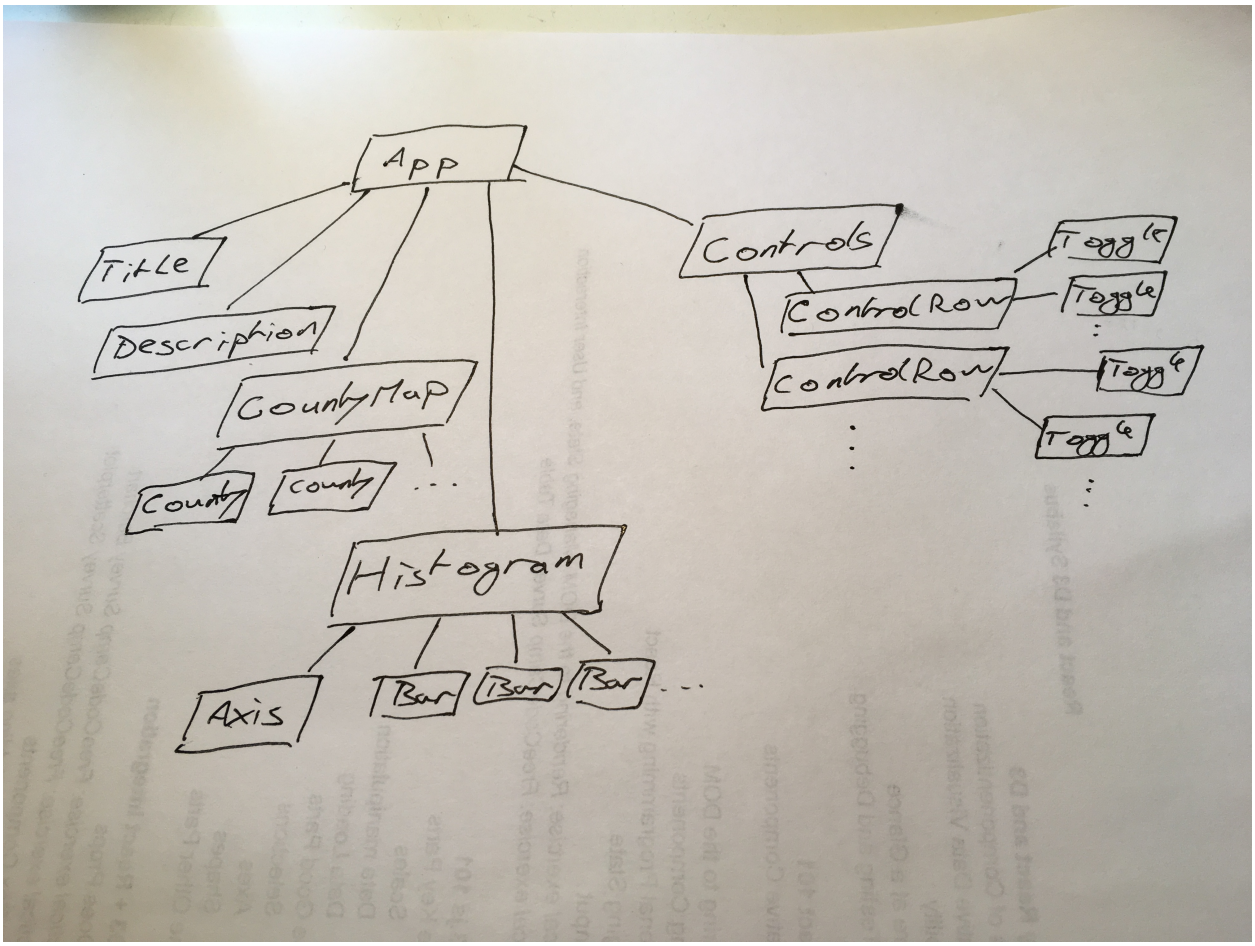
I've shot myself in the foot *a lot*. Life gets harder and harder until one day you want to throw all your code away and embark on The Rewrite. That's how projects die.

The Rewrite killed Netscape[13]. You probably don't even remember Netscape :wink:

Let's save you from that.

[13] http://www.joelonsoftware.com/articles/fog0000000069.html

Basic architecture



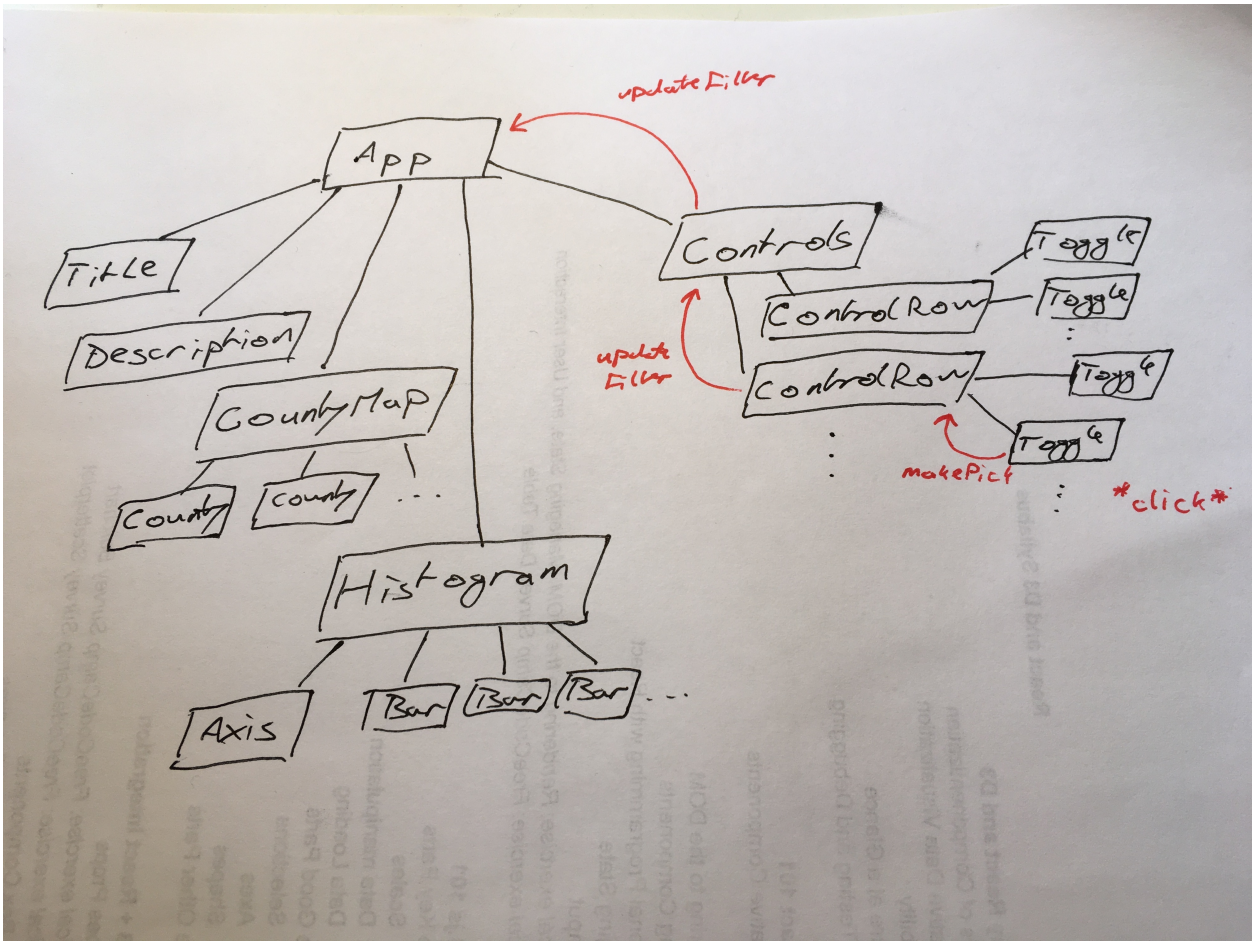
The basic architecture

We're going to use a unidirectional dataflow architecture inspired by Flux:

- The Main Component – `App` – is the repository of truth
- Child components react to user events
- They announce changes using callbacks
- The Main Component updates its truth
- The real changes flow back down the chain to update UI

This looks roundabout, but it's awesome. It's far better than worrying about parts of the UI going out of date with the rest of the app. I could talk your ear off with debugging horror stories, but I'm nice, so I won't.

When a user clicks on one of our controls, a `Toggle`, it invokes a callback. This in turn invokes a callback on `ControlRow`, which invokes a callback on `Controls`, which invokes a callback on `App`.

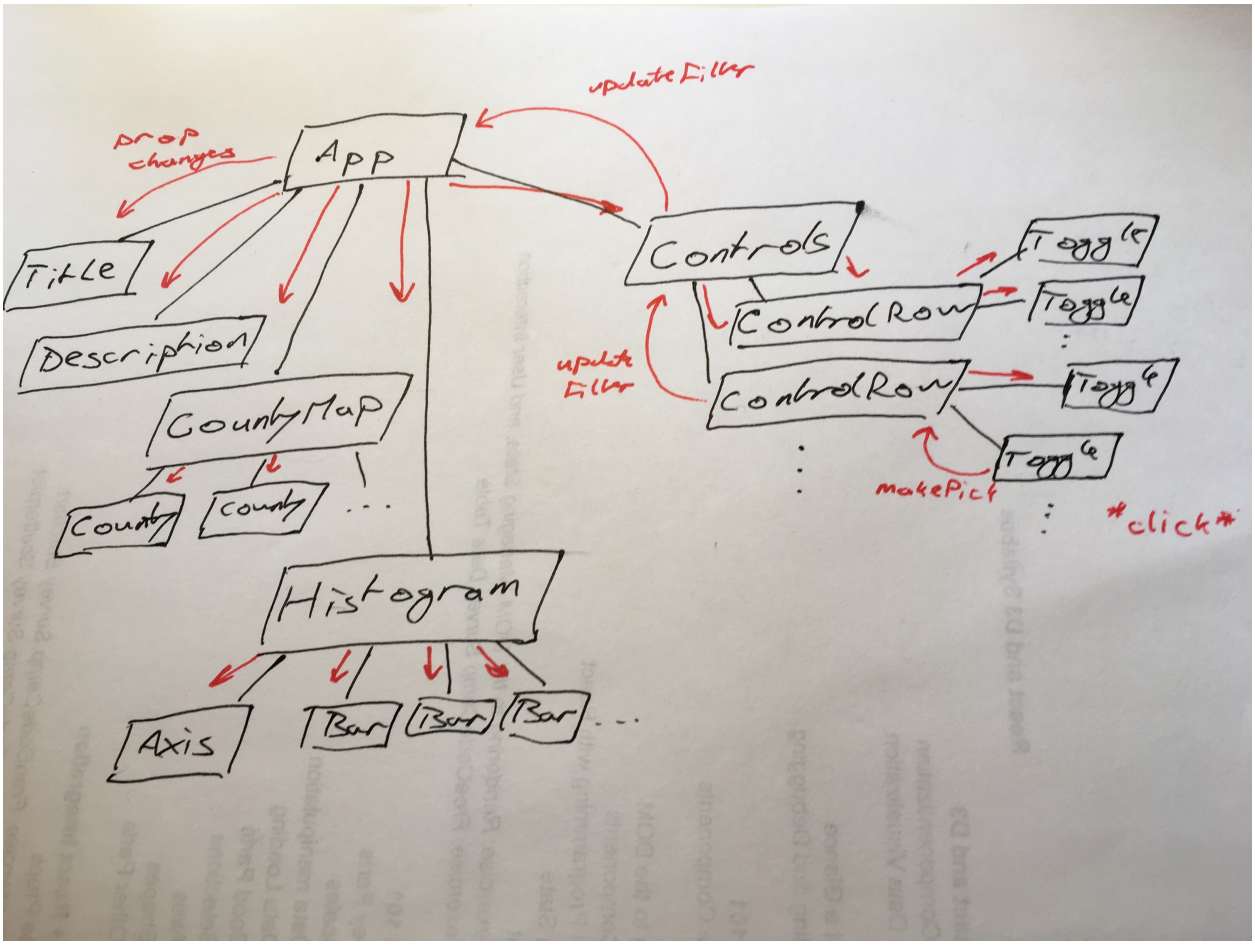


Callback chain

With each hop, the nature of our callback changes. `Toggle` tells `ControlRow` which entry was toggled, `ControlRow` tells `Controls` how to update the data filter function, and `Controls` gives `App` a composite filter built from all the controls. You'll see how that works in the next chapter.

The important takeaway right now is that callbacks go from low-level info to high-level semantic info.

When the final callback is invoked, `App` updates its repository of truth and communicates the change back down the chain via props. This happens with no additional wiring on your part.



Data flows down

You can think of it like calling functions with new arguments. Because the functions – components – render the UI, your interface updates.

Because your components are well-made and rely on their props to render, React's engine can optimize these changes. It can compare the component tree and decide which components to re-render and which to leave alone.

Functional programming for HTML! :sunglasses:

The functional programming concepts we're relying on are called referential transparency[14], idempotent functions[15], and functional purity[16]. I suggest Googling them if you want to learn the theory behind it all.

[14] https://en.wikipedia.org/wiki/Referential_transparency
[15] https://en.wikipedia.org/wiki/Idempotence
[16] https://en.wikipedia.org/wiki/Pure_function

A caveat

The caveat with our basic state handling approach is that it's less flexible and scalable than using a state handling library like Redux or MobX. The more components we add, and the more interaction we come up with, the harder it will become to keep track of our callbacks. Redesigning the UI is also cumbersome because you have to rewire all the callbacks.

We're using the basic approach because it's easier to explain, works without additional libraries, and is Good Enough™. I mention Redux and MobX to make your Googling easier.

You can see an approach to using Redux in dataviz in the Animating with React, Redux, and D3 chapter, and we'll tackle MobX in the MobX chapter.

Structuring your React app

We're going to structure our app into components. Deciding what to put into one component and what to put into another is one of the hardest problems in engineering.

Entire books have been written on the topic, but here's a rule of thumb that I like to use: if you have to use the word "and" to describe what your component does, then it should become two components.

Once you have those two components, you can either make them child components of a bigger component, or you can make them separate. The choice depends on their re-usability and often mimics your design structure.

For example, our tech salary visualization is going to use 1 very top level component, 5 major components, and a bunch of child components.

- `App` is the very top level component; it handles everything
  - `Title` renders the dynamic title
  - `Description` renders the dynamic description
  - `Histogram` renders the histogram and has child components for the axis and histogram bars
  - `CountyMap` renders the choropleth map and uses child components for the counties
  - `Controls` renders the rows of buttons that let users explore our dataset

Most of these are specific to our use case, but `Histogram` and `CountyMap` have potential to be used elsewhere. We'll keep that mind when we build them.

`Histogram`, `CountyMap`, and `Controls` are going to have their own folder inside `src/components/` to help us group major components with their children. An `index.js` file will help with imports.

We'll use a `Meta` folder for all our metadata components like `Title` and `Description`. We don't *have* to do this, but `import { Title, Description } from './Meta'` looks better than doing separate imports for related-but-different components. Namespacing, if you will.

Each component should be accessible with `import My Component from './MyComponent'` and rendered with `<MyComponent {...params} />`. If a parent component has to know details about the implementation of a child component, something is wrong.

You can read more about these ideas by Googling "leaky abstractions"[17], "single responsibility principle"[18], "separation of concerns"[19], and "structured programming"[20]. Books from the late 90's and early 2000's (when object-oriented programming was The Future™) are the best source of curated info in my experience.

[17] https://en.wikipedia.org/wiki/Leaky_abstraction
[18] https://en.wikipedia.org/wiki/Single_responsibility_principle
[19] https://en.wikipedia.org/wiki/Separation_of_concerns
[20] https://en.wikipedia.org/wiki/Structured_programming

—

Congratz! You know everything you need to build visualizations with React and D3. :clap:

This is the point in tech books where I run off and start building things on my own. Then I get frustrated, spend hours Googling for answers, and then remember, "Hey! Maybe I should read the rest of the book!"

Reading the rest of the book helps. I'll show you how all this stuff fits together into a larger project.

Set up a local environment with create-react-app

Can you believe this is the third time I'm writing a *"How to set up all the tools and get started"* section? The ecosystem moves fast!

You can see the old versions as Appendix A – a roll-your-own environment that's almost the same as the one I'm about to show you, and Appendix B – a browserify-based environment (if you don't like Webpack).

In the summer of 2016, the React team released a tool called `create-react-app`. It creates a React app so that you can get started right away. All of the code in this book runs with an app set up like this.

Getting started with React has never been easier.

1. Make sure you have node.js
2. Install the app generator
3. Run the app generator

Make sure you have node.js

Modern JavaScript development runs on node.js. Your code still ends up in a browser, but there are a few steps it has to go through before it gets there. Those toolchains run on node.js.

Go to nodejs.org[21], download one of the latest versions, and run the installer. You can pick the more stable LTS (long-term-support) version, or the more gimme-all-the-features version. JavaScript toolchains run fine in both.

Server-side rendering might be easier with the fancy-pants version. More on that later.

Install create-react-app

Great, you have node and you're ready to go.

Run this command in a terminal:

[21] https://nodejs.org/en/

Install create-react-app

```
$ npm install -g create-react-app
```

This installs a global npm package called `create-react-app`. Its job is to create a directory and install `react-scripts`.

Confusing, I know.

You can think of `create-react-app` and `react-scripts` as two parts of the same construction. The first is a lightweight script that you can install globally and never update; the second is where the work happens. You want this one to be fresh every time you start a new app.

Keeping a global dependency up to date would suck, especially considering there have been six major updates since Facebook first launched `create-react-app` in July 2016.

Run create-react-app

Superb! You have `create-react-app`. Time to create an app and get started with some coding.

Run this in a terminal:

Create your project

```
$ create-react-app react-d3js-example
```

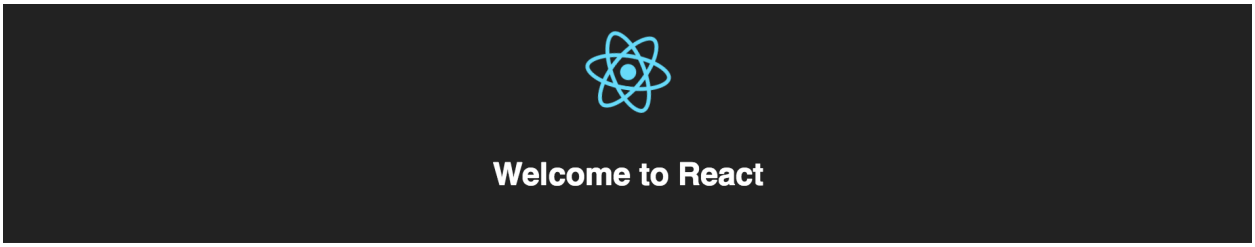
Congratulations! You just created a React app. *With* all the setup and the fuss that goes into using a modern JavaScript toolchain.

Your next step is to run your app:

Start your dev server

```
$ cd react-d3js-example
$ npm start
```

A browser tab should open with a page that looks like this:



Initial React app

If that didn't work, then something must have gone terribly wrong. You should consult the official docs[22]. Maybe that will help.

What you get

Running `create-react-app` installs tools and libraries. There's around 80MB of them as of October 2016. This is why using a generator is easier than slogging through on your own.

Crucially, there is a single dependency in your project – `react-scripts`. But it gives you everything you need to build modern React apps.

- **Webpack** - a module bundler and file loader. It turns your app into a single JavaScript file, and it even lets you import images and styles like they were code.
- **Babel** - a JavaScript transpiler. It turns your modern JS code (ES6, ECMAScript2015, 2016, ES7, whatever you call it) into code that can run on real world browsers. It's the ecosystem's answer to slow browser adoption.
- **ESLint** - linting! It annoys you when you write code that is bad. This is a good thing. :smile:

[22] https://github.com/facebookincubator/create-react-app

- **Jest** - a test runner. Having tests set up from the beginning of a project is a good idea. We won't really write tests in this book, but I will show you how it's done.

All tools come preconfigured with sane defaults and ready to use. You have no idea how revolutionary this is. Check the appendixes to see how hard setting up an environment used to be. I'm so happy that `create-react-app` exists.

—

Besides the toolchain, `create-react-app` also gives you a default directory structure and a ton of helpful docs.

You should read the `README.md` file. It's full of great advice, and it goes into more detail about the app generator than we can get into here. The team also makes sure it's kept up-to-date.

All of the code samples in this book are going to follow the default directory structure. Public assets like HTML files, datasets, and some images go into `public/`. Code goes into `src/`.

Code is anything that we `import` or `require()`. That includes JavaScript files themselves, as well as stylesheets and images.

You can poke around `src/App.js` to see how it's structured and what happens when you change something. You're going to change bits and pieces of this file as you go through the book.

I suggest you keep `npm start` running at all times while coding. The browser will refresh on every code change, so you'll be able to see instant results.

Install dependencies for this book

There are a couple of libraries we're going to use often in this book: D3, Topojson, and Lodash. We're using D3 to do our heavy lifting, Topojson to parse geographical data, and Lodash to make data manipulation easier.

You can install them like this:

Install dependencies

```
1 $ npm install --save d3 topojson lodash
```

Additionally, we're using Bootstrap for default styling and String for string manipulation. You should install them as well.

Styling and string manipulation

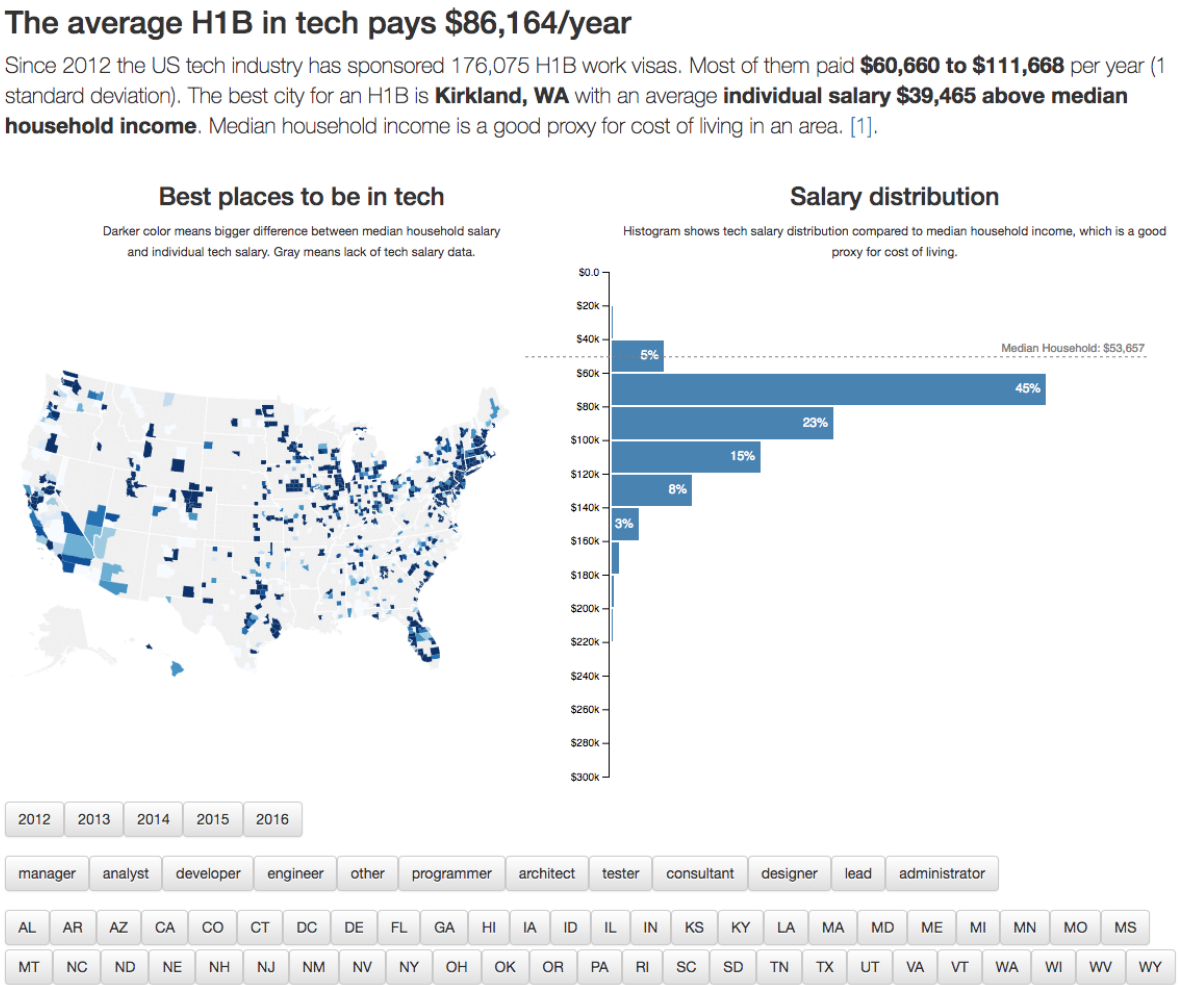
```
$ npm install --save bootstrap string
```

Now your environment is ready! You should see a default React App page in the browser without errors. Keep `npm start` running as you follow the examples in this book.

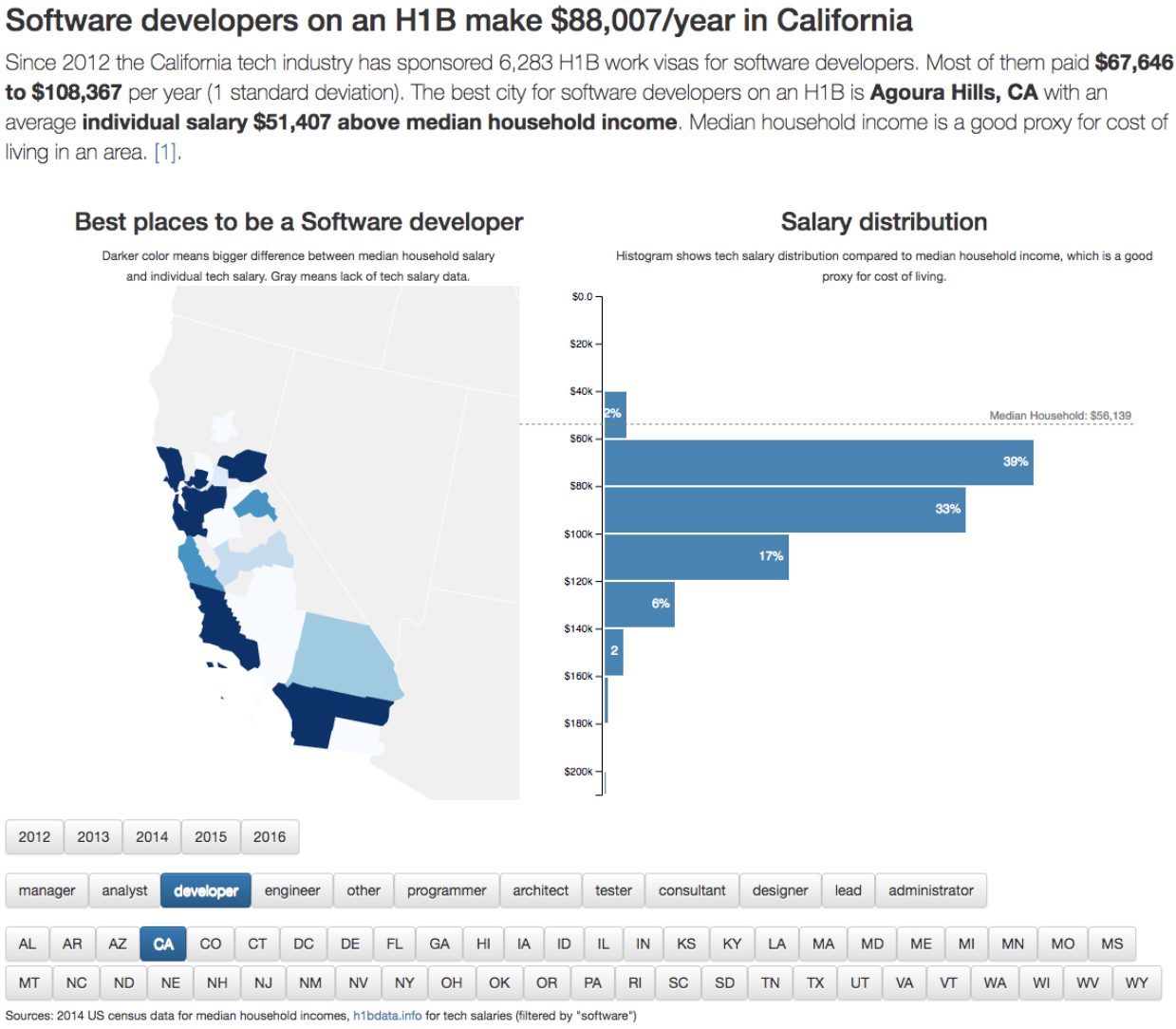
Some examples are going to need other libraries. When that happens, I'll remind you to install them.

A big example project - 176,113 tech salaries visualized

We're going to build this:



It's an interactive visualization app with a choropleth map and a histogram comparing tech salaries with median household income in the area. Users can filter by three different parameters – year, job title, and US state – to get a more detailed view.



It's going to be great.

At this point, I assume you've used `create-react-app` to set up your environment. Check the getting started section if you haven't. I'll also assume you've read the basics chapter. I'm still going to explain what we're doing, but knowing the basics helps.

I suggest you follow along, keep `npm start` running, and watch your visualization change in real time as you code. It's rewarding as hell.

If you get stuck, you can use my react-d3js-step-by-step Github repo[23] to jump between steps. The 9 tags[24] correspond to the code at the end of each step. Download the first tag and run `npm install` to skip the initial setup.

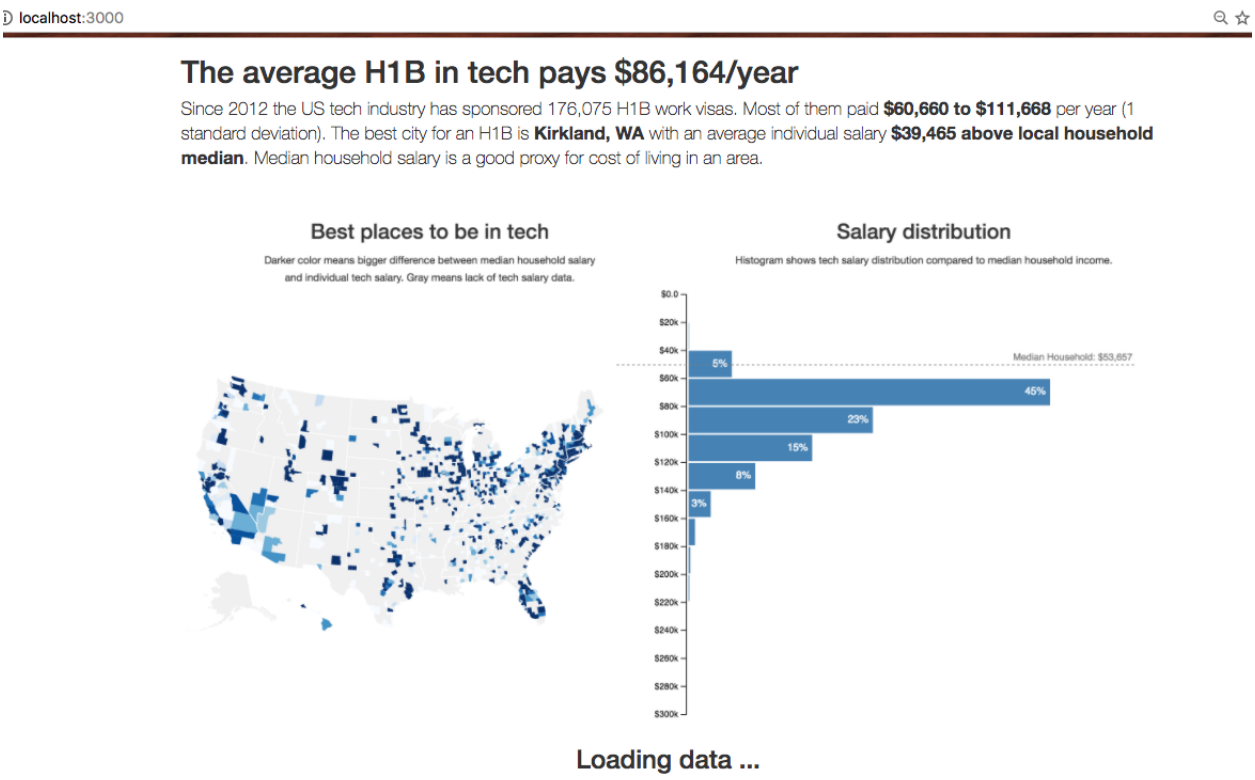
[23] https://github.com/Swizec/react-d3js-step-by-step
[24] https://github.com/Swizec/react-d3js-step-by-step/releases

If you want to see how this project evolved over 22 months, check the original repo[25]. The create-react-app[26] branch has the code you're about to build.

[25] https://github.com/Swizec/h1b-software-salaries

[26] https://github.com/Swizec/h1b-software-salaries/tree/create-react-app

Show a Preloader



Preloader screenshot

Our preloader is a screenshot of the final result. Usually you'd have to wait until the end of the project to make that, but I'll just give you mine. Starting with the preloader makes sense for two reasons:

1. It's nicer than looking at a blank screen while data loads
2. It's a good sanity check for our environment

We're using a screenshot of the final result because the full dataset takes a few seconds to load, parse, and render. It looks better if visitors see something informative while they wait.

Make sure you've installed all dependencies and that `npm start` is running.

We're building the preloader in 4 steps:

1. Get the image
2. Make the `Preloader` component
3. Update `App`
4. Load Bootstrap styles in `index.js`

Step 1: Get the image

Download my screenshot from Github[27] and save it in `src/assets/preloading.png`. It goes in the `src/assets/` directory because we're going to `import` it in JavaScript (which makes it part of our source code), and I like to put non-JavaScript files in `assets`. It keeps the project organized.

Step 2: Preloader component

The `Preloader` is a component that pretends it's the `App` and renders a static title, description, and a screenshot of our end result. It goes in `src/components/Preloader.js`.

We'll put all of our components in `src/components/`.

We start the component off with some imports, an export, and a functional stateless component that returns an empty div element.

Preloader skeleton

```
1  // src/components/Preloader.js
2  import React from 'react';
3
4  import PreloaderImg from '../assets/preloading.png';
5
6  const Preloader = () => (
7      <div className="App container">
8
9      </div>
10 );
11
12 export default Preloader;
```

We `import` React (which we need to make JSX syntax work) and the `PreloaderImg` for our image. We can import images because of the Webpack configuration that comes with `create-react-app`. The image loader puts a file path in the `PreloaderImg` constant.

[27] https://raw.githubusercontent.com/Swizec/react-d3js-step-by-step/798ec9eca54333da63b91c66b93339565d6d582a/src/assets/preloading.png

At the bottom, we `export default Preloader` so that we can use it in `App.js` as `import Preloader`. I like to use default exports when the file exports a single thing and named exports when we have multiple. You'll see that play out in the rest of this project.

The `Preloader` function takes no props (because we don't need any) and returns an empty `div`. Let's fill it in.

Preloader content

```
// src/components/Preloader.js
const Preloader = () => (
    <div className="App container">
        <h1>The average H1B in tech pays $86,164/year</h1>
        <p className="lead">
            Since 2012 the US tech industry has sponsored 176,075 H1B work visas.
            Most of them paid <b>$60,660 to $111,668</b> per year (1 standard de\
viation).
            <span>
                The best city for an H1B is <b>Kirkland, WA</b> with an average \
individual
                salary <b>$39,465 above local household median</b>.
                Median household salary is a good proxy for cost of living in an\
 area
            </span>
        </p>
        <img src={PreloaderImg} style={{width: '100%'}} role="presentation" />
        <h2 className="text-center">Loading data ...</h2>
    </div>
);
```

We're cheating again because I copy-pasted that from the finished example. You wouldn't have anywhere to get this yet.

The code itself looks like plain HTML. We have the usual tags - `h1`, `p`, `b`, `img`, and `h2`. That's what I like about using JSX. It feels familiar.

But look at the `img` tag: the `src` attribute is dynamic, defined by `PreloaderImg`, and the `style` attribute takes an object, not a string. That's because JSX is more than HTML; it's JavaScript. You can put any JavaScript entity you need in those props.

That will be one of the cornerstones of our sample project.

Step 3: Update App

To use our new `Preloader` component, we have to edit `src/App.js`. Let's start by removing the defaults that came with `create-react-app` and importing our `Preloader` component.

Revamp App.js

```
// src/App.js
import React, { Component } from 'react';
import logo from './logo.svg';
import './App.css';

import Preloader from './components/Preloader';

class App extends Component {
  render() {
    return (
      <div className="App">
        <div className="App-header">
          <img src={logo} className="App-logo" alt="logo" />
          <h2>Welcome to React</h2>
        </div>
        <p className="App-intro">
          To get started, edit <code>src/App.js</code> and save to reload.
        </p>
      </div>
    );
  }
}

export default App;
```

We removed the logo and style imports, added an import for `Preloader`, and gutted everything out of the `App` class. It's a great starting point for a default app, but it's served its purpose.

Let's define a default `state` and a `render` method that uses our `Preloader` component when there's no data.

Render the preloader

```
// src/App.js
class App extends Component {
    state = {
        techSalaries: []
    }

    render() {
        if (this.state.techSalaries.length < 1) {
```

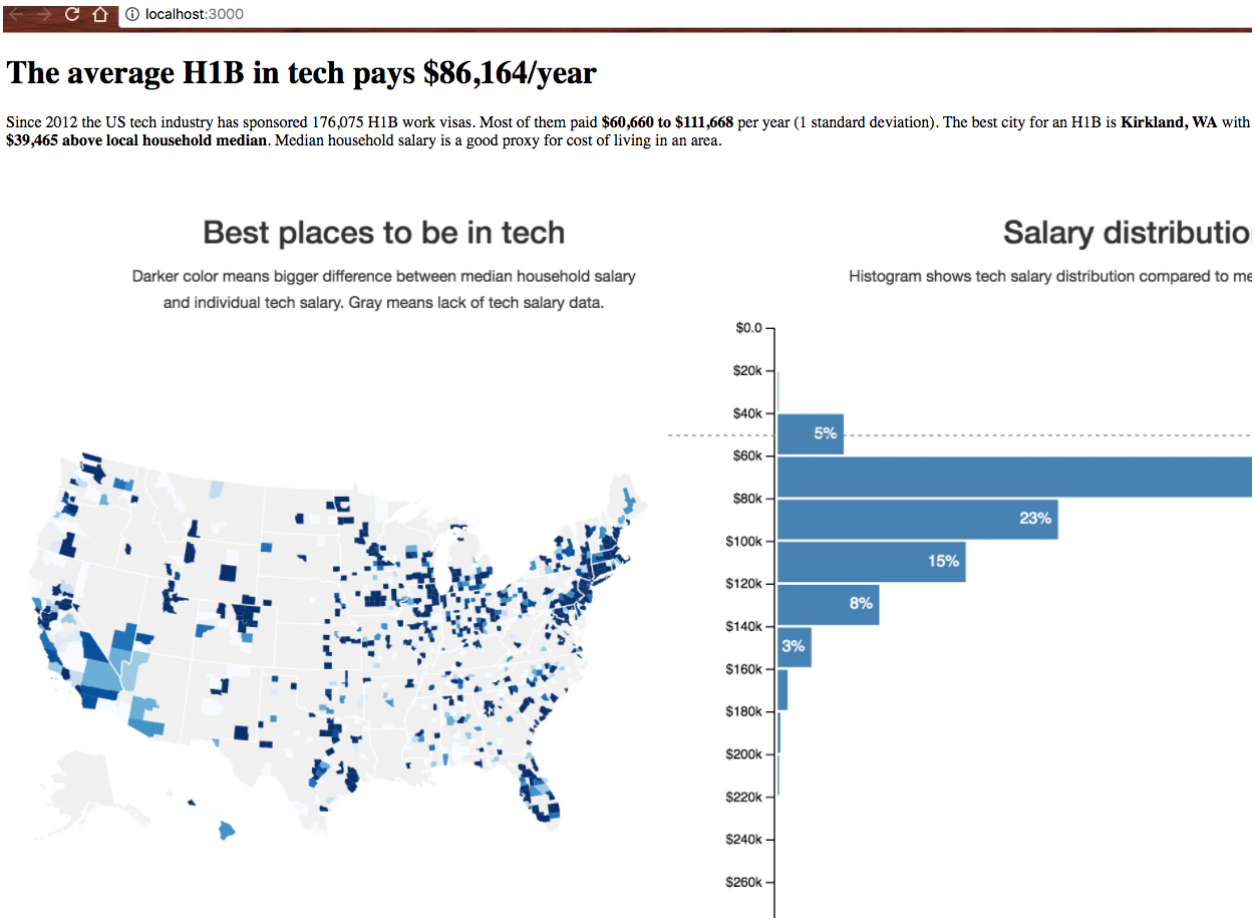
```
            return (
                <Preloader />
            );
        }

        return (
            <div className="App">

            </div>
        );
    }
}
```

With modern ES6+ classes, we can define properties directly in the class without going through the constructor method. This makes our code cleaner and easier to read.

You might be wondering whether `state` is now a class static property or if it's bound to `this` for each object. It works the way we need it to: bound to each `this` instance. I don't know *why* it works that way because it's hard to Google for these things when you can't remember the name, but I know it works. Tried and battle tested :smile:

We set `techSalaries` to an empty array, then in `render` check whether it's empty and render either the `Preloader` component or a blank `<div>`. Rendering your preloaders when there's no data makes sense even if you still need to build your data loading.

If you have `npm start` running, your preloader should show up on screen.



Preloader without Bootstrap styles

Hmm… that's not very pretty. Let's fix it.

Step 4: Load Bootstrap styles

We're going to use Bootstrap styles to avoid reinventing the wheel. We're ignoring their JavaScript widgets and the amazing integration built by the react-bootstrap[28] team. All we need are the stylesheets.

They'll make the fonts look better, help with layouting, and make buttons look like buttons.

We add them in `src/index.js`.

[28] http://react-bootstrap.github.io/

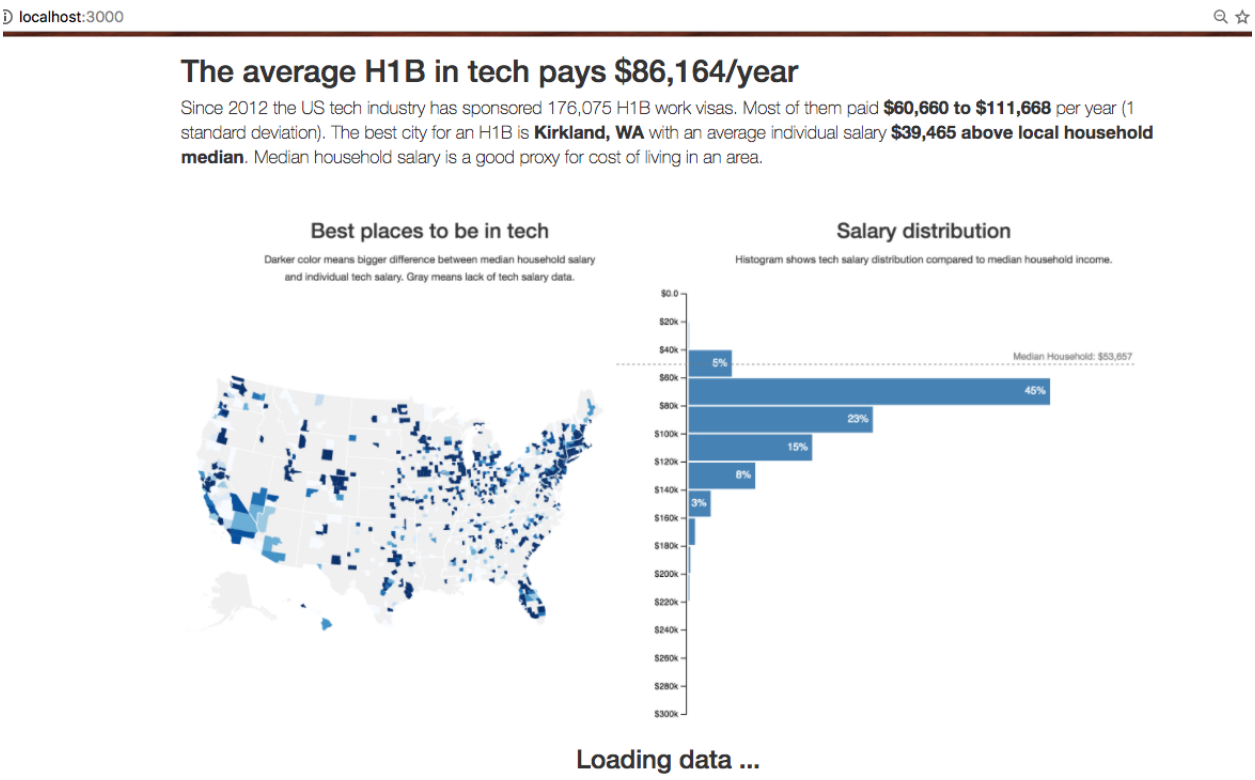
Add bootstrap in index.js

```
// src/index.js
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';
import 'bootstrap/dist/css/bootstrap.css';
import 'bootstrap/dist/css/bootstrap-theme.css';

ReactDOM.render(
  <App />,
  document.getElementById('root')
);
```

Another benefit of using Webpack: `import`-ing stylesheets. These imports turn into `<style>` tags with CSS in their body at runtime.

This is also a good opportunity to see how simple the `index.js` file is. It requires our `App` and uses `ReactDOM` to render it into the page. That's it.

You should now see your beautiful preloader on screen.



Preloader screenshot

If you don't, try comparing your changes to this diff on Github[29].

[29] https://github.com/Swizec/react-d3js-step-by-step/commit/798ec9eca54333da63b91c66b93339565d6d582a

Asynchronously load data

Great! We have a preloader. Time to load some data.

We'll use D3's built-in data loading methods and tie their callbacks into React's component lifecycle. You could talk to a REST API in the same way. Neither D3 nor React care what the datasource is.

First, you need the data files. I scraped the tech salary info from h1bdata.info[30], the median household incomes from the US census datasets, and US map data from Mike Bostock's github repositories. I used some elbow grease and python scripts to tie the datasets together.

You can read about the scraping on my blog here[31], here[32], and here[33]. But it's not the subject of this book.

Step 0: Get the data

You should download the 6 datafiles from the step-by-step repository on Github[34]. Put them in the `public/data` directory in your project.

First, you need the data files. I scraped the tech salary info from h1bdata.info[35], the median household incomes from the US census datasets, and US map data from Mike Bostock's Github repositories. I used some elbow grease and python scripts to tie the datasets together.

You can read about the scraping on my blog here[36], here[37], and here[38]. But it's not the subject of this book.

You should download the 6 data files from the step-by-step repository on Github[39]. Put them in the `public/data` directory in your project.

The quickest way to download each file is to click `View`, then right-click `Raw` and `Save Link As`.

[30] http://h1bdata.info/
[31] https://swizec.com/blog/place-names-county-names-geonames/swizec/7083
[32] https://swizec.com/blog/facts-us-household-income/swizec/7075
[33] https://swizec.com/blog/livecoding-24-choropleth-react-js/swizec/7078
[34] https://github.com/Swizec/react-d3js-step-by-step/commit/8819d9c38b4aef0a0c569e493f088ff9c3bfdf33
[35] http://h1bdata.info/
[36] https://swizec.com/blog/place-names-county-names-geonames/swizec/7083
[37] https://swizec.com/blog/facts-us-household-income/swizec/7075
[38] https://swizec.com/blog/livecoding-24-choropleth-react-js/swizec/7078
[39] https://github.com/Swizec/react-d3js-step-by-step/commit/8819d9c38b4aef0a0c569e493f088ff9c3bfdf33

Step 1: Prep App.js

Let's set up our `App` component first. That way you'll see results as soon data loading starts to work.

We start by importing our data loading method - `loadAllData` - and both D3 and Lodash. We'll need them later.

Import d3, lodash, and our data loader

```
// src/App.js
import React, { Component } from 'react';
import * as d3 from 'd3';
import _ from 'lodash';

import Preloader from './components/Preloader';
import { loadAllData } from './DataHandling';
```

You already know the normal imports. Importing with `{}` is how we import named exports, which lets us get multiple things from the same file. You'll see how the export side works in Step 2.

Initiate data loading in App.js

```
// src/App.js
class App extends Component {
    state = {
        techSalaries: [],
        countyNames: [],
        medianIncomes: []
    }

    componentWillMount() {
        loadAllData(data => this.setState(data));
    }
```

We initiate data loading inside the `App` class's `componentWillMount` lifecycle hook. It fires right before React mounts our component into the DOM. Seems like a good place to start loading data, but some say it's an anti-pattern.

I like tying it to component mount when using the basic architecture, and in a more render agnostic place when using Redux or MobX for state management.

To initiate data loading, we call the `loadAllData` function, which we're defining next, then use `this.setState` in a callback. This updates `App`'s state and triggers a re-render, which updates our entire visualization via props.

We also took this opportunity to add two more entries to our `state`: `countyNames`, and `medianIncomes`.

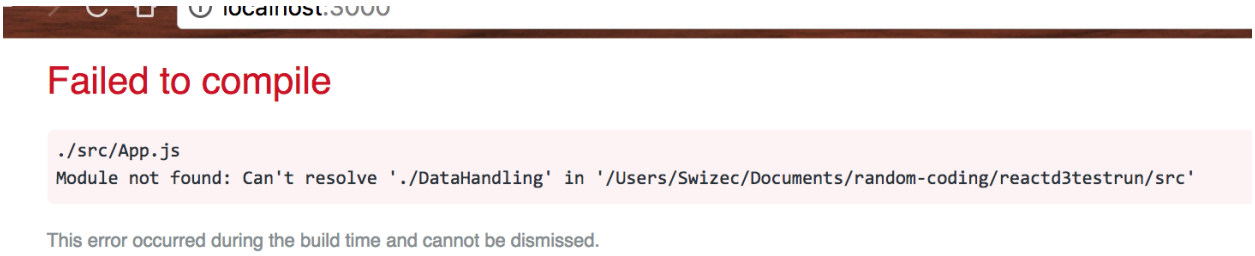
Let's add a "Data loaded" indicator to the `render` method. That way we'll know when data loading works.

Data loaded indicator

```
    // src/App.js
    render() {
         if (this.state.techSalaries.length < 1) {
             return (
                 <Preloader />
             );
          }

        return (
            <div className="App">
            <div className="App container">
                <h1>Loaded {this.state.techSalaries.length} salaries</h1>
            </div>
        );
    }
```

We added the `container` class to the main `<div>` and added an `<h1>` tag to show how many datapoints were loaded. The `{}` pattern denotes a dynamic value in JSX. You've seen this in props so far, but it works in tag bodies as well.

With all of this done, you should see an error overlay.



DataHandling.js not found error overlay

These nice error overlays come with `create-react-app`. They make it so you never have to check the terminal where `npm start` is running. A big improvement thanks to the React team at Facebook.

Let's build that file and fill it with our data loading logic.

Step 2: Prep data parsing functions

We're putting data loading logic in a file separate from `App.js` because it's a bunch of functions that work together and don't have much to do with the `App` component.

We start the file with two imports and four data parsing functions:

- `cleanIncomes`, which parses each row of household income data
- `dateParse`, which we use for parsing dates
- `cleanSalary`, which parses each row of salary data
- `cleanUSStateName`, which parses US state names

Data parsing functions

```
// src/DataHandling.js
import * as d3 from 'd3';
import _ from 'lodash';

const cleanIncomes = (d) => ({
    countyName: d['Name'],
    USstate: d['State'],
    medianIncome: Number(d['Median Household Income']),
    lowerBound: Number(d['90% CI Lower Bound']),
    upperBound: Number(d['90% CI Upper Bound'])
});

const dateParse = d3.timeParse("%m/%d/%Y");

const cleanSalary = (d) => {
    if (!d['base salary'] || Number(d['base salary']) > 300000) {
        return null;
    }

    return {employer: d.employer,
            submit_date: dateParse(d['submit date']),
            start_date: dateParse(d['start date']),
            case_status: d['case status'],
            job_title: d['job title'],
            clean_job_title: d['job title'],
            base_salary: Number(d['base salary']),
            city: d['city'],
            USstate: d['state'],
            county: d['county'],
            countyID: d['countyID']
```

```
    };
}

const cleanUSStateName = (d) => ({
    code: d.code,
    id: Number(d.id),
    name: d.name
});
```

You'll see those `d3` and `lodash` imports a lot.

The data parsing functions all follow the same approach: Take a row of data as `d`, return a dictionary with nicer key names, and cast any numbers or dates into appropriate formats. They all come in as strings.

Doing the parsing and the nicer key names now makes the rest of our codebase simpler because we don't have to deal with this all the time. For example, `entry.job_title` is nicer to read and type than `entry['job title']`.

Step 3: Load the datasets

Now that we have our data parsing functions, we can use D3 to load the data with Ajax requests.

Data loading

```
// src/DataHandling.js
export const loadAllData = (callback = _.noop) => {
    d3.queue()
      .defer(d3.json, 'data/us.json')
      .defer(d3.csv, 'data/us-county-names-normalized.csv')
      .defer(d3.csv, 'data/county-median-incomes.csv', cleanIncomes)
      .defer(d3.csv, 'data/h1bs-2012-2016-shortened.csv', cleanSalary)
      .defer(d3.tsv, 'data/us-state-names.tsv', cleanUSStateName)
      .await((error, us, countyNames, medianIncomes, techSalaries, USstateNames)\
 => {

      });
};
```

Here you can see another ES6 trick: default argument values. If `callback` is false, we set it to `_.noop` - a function that does nothing. This lets us later call `callback()` without worrying whether it was given as an argument.

`d3.queue` lets us call multiple asynchronous functions and wait for them all to finish. By default, it runs all functions in parallel, but that's configurable through an argument - `d3.queue(1)` for one at a time, `2` for two, etc. In our case, without an argument, it runs all tasks in parallel.

We define `5` tasks to run with `.defer` then wait for them to finish with `.await`. The tasks themselves are D3's data loading functions that fire an Ajax request to the specified URL, parse the data into a JavaScript dictionary, and use the given row parsing function to polish the result.

For instance, `.defer(d3.csv, 'data/county-median-incomes.csv', cleanIncomes)` uses `d3.csv` to make an Ajax request to `data/county-median-incomes.csv`, parses the CSV file into an array of JavaScript dictionaries, and uses `cleanIncomes` to further parse each row the way we specified earlier.

D3 supports formats like `json`, `csv`, `tsv`, `text`, and `xml` out of the box. You can make it work with custom data sources through the underlying `request` API.

PS: we're using the shortened salary dataset to make page reloads faster while building our project.

Step 4: Tie the datasets together

If you put a `console.log` in the `.await` callback above, you'll see a bunch of data. Each argument - `us`, `countyNames`, `medianIncomes`, `techSalaries`, `USstateNames` - holds the entire parsed dataset from the corresponding file.

To tie them together and prepare a dictionary for `setState` back in the `App` component, we need to add a little big of logic. We're going to build a dictionary of county household incomes and remove any empty salaries.

Tie the datasets together

```
// src/DataHandling.js
.await((error, us, countyNames, medianIncomes, techSalaries, USstateNames) => {
    countyNames = countyNames.map(({ id, name }) => ({id: Number(id),
                                                      name: name}));

    let medianIncomesMap = {};

    medianIncomes.filter(d => _.find(countyNames,
                                     {name: d['countyName']}))
                 .forEach((d) => {
                     d['countyID'] = _.find(countyNames,
                                            {name: d['countyName']}).id;
                     medianIncomesMap[d.countyID] = d;
                 });

```

```
    techSalaries = techSalaries.filter(d => !_.isNull(d));

    callback({
        usTopoJson: us,
        countyNames: countyNames,
        medianIncomes: medianIncomesMap,
        medianIncomesByCounty: _.groupBy(medianIncomes, 'countyName'),
        medianIncomesByUSState: _.groupBy(medianIncomes, 'USstate'),
        techSalaries: techSalaries,
        USstateNames: USstateNames
    });
});
```

The first line should be one of those `cleanX` functions like we had above. I'm not sure how I missed it.

Then we have the county median income map building. It looks like weird code because of the indentation, but it's not that bad. We `filter` the `medianIncomes` array to discard any incomes whose `countyName` we can't find. I made sure all the names are unique when I was building the datasets.

We use `forEach` to walk through the filtered array, find the correct `countyID`, and add the entry to `medianIncomesMap`. When we're done, we have a large dictionary that maps county ids to their household income data.

At the end, we filter `techSalaries` to remove any empty values - the `cleanSalaries` function returns `null` when a salary is either undefined or absurdly high.

Then we call `callback` with a dictionary of the new datasets. To make future access quicker, we use `_.groupBy` to build dictionary maps of counties by county name and by US state.

You should now see how many salary entries the shortened dataset contains.

Data loaded screenshot

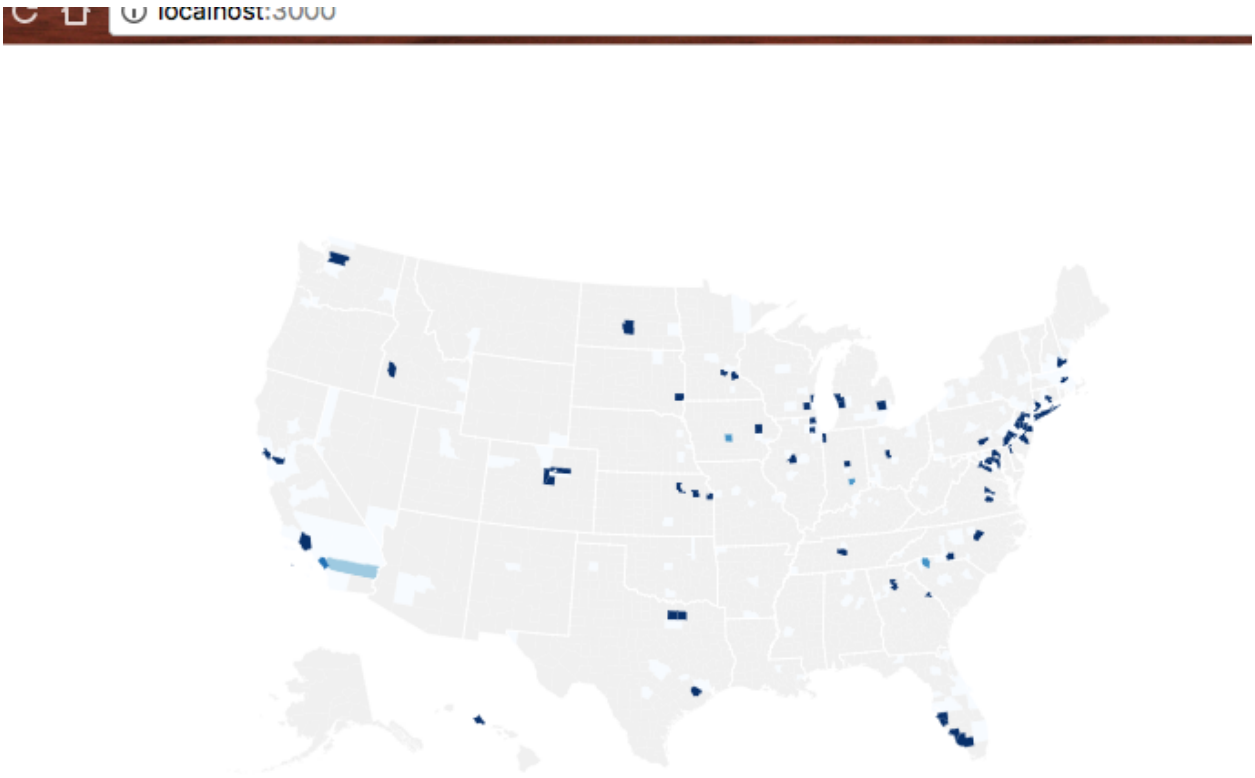
If that didn't work, try comparing your changes to this diff on Github[40].

[40] https://github.com/Swizec/react-d3js-step-by-step/commit/9f113cdd3bc18535680cb5a4e87a3fd45743c9ae

Render a choropleth map of the US

Now that we have our data, it's time to start drawing pictures - a choropleth map. That's a map that uses colored geographical areas to encode data.

In this case, we're going to show the delta between median household salary in a statistical county and the average salary of a single tech worker on a visa. The darker the blue, the higher the difference.



Choropleth map with shortened dataset

There's a lot of gray on this map because the shortened dataset doesn't span that many counties. There's going to be plenty in the full choropleth too, but not as much as there is here.

Turns Out™ immigration visa opportunities for techies aren't evenly distributed throughout the country. Who knew?

Just like before, we're going to start with changes in our `App` component, then build the new bit. This time, a `CountyMap` component spread into three files:

- `CountyMap/index.js`, to make imports easier
- `CountyMap/CountyMap.js`, for overall map logic
- `CountyMap/County.js`, for individual county polygons

Step 1: Prep App.js

You might guess the pattern already: add an import, add a helper method or two, update `render`.

Import CountyMap component

```
// src/App.js
import Preloader from './components/Preloader';
import { loadAllData } from './DataHandling';

import CountyMap from './components/CountyMap';
```

That imports the `CountyMap` component from `components/CountyMap/`. Until we're done, your browser should show an error overlay about some file or another not existing.

In the `App` class itself, we add a `countyValue` method. It takes a county entry and a map of tech salaries, and it returns the delta between median household income and a single tech salary.

App.countyValue method

```
    // src/App.js
    countyValue(county, techSalariesMap) {
        const medianHousehold = this.state.medianIncomes[county.id],
              salaries = techSalariesMap[county.name];

        if (!medianHousehold || !salaries) {
            return null;
        }

        const median = d3.median(salaries, d => d.base_salary);

        return {
            countyID: county.id,
            value: median - medianHousehold.medianIncome
        };
    }
```

We use `this.state.medianIncomes` to get the median household salary and the `techSalariesMap` input to get salaries for a specific census area. Then we use `d3.median` to calculate the median value for salaries and return a two-element dictionary with the result.

`countyID` specifies the county and `value` is the delta that our choropleth displays.

In the `render` method, we'll do three things:

- prep a list of county values
- remove the "data loaded" indicator
- render the map

Render the CountyMap component

```
    // src/App.js
    render() {
        if (this.state.techSalaries.length < 1) {
            return (
                <Preloader />
            );
        }

        const filteredSalaries = this.state.techSalaries,
              filteredSalariesMap = _.groupBy(filteredSalaries, 'countyID'),
              countyValues = this.state.countyNames.map(
                  county => this.countyValue(county, filteredSalariesMap)
              ).filter(d => !_.isNull(d));

        let zoom = null;

          return (
              <div className="App container">
                  <h1>Loaded {this.state.techSalaries.length} salaries</h1>
                  <svg width="1100" height="500">
                      <CountyMap usTopoJson={this.state.usTopoJson}
                                 USstateNames={this.state.USstateNames}
                                 values={countyValues}
                                 x={0}
                                 y={0}
                                 width={500}
                                 height={500}
                                 zoom={zoom} />
                  </svg>
              </div>
          );
      }
```

We call our dataset `filteredTechSalaries` because we're going to add filtering in the subchapter about adding user controls. Using the proper name now means less code to change later. The magic of foresight :smile:

We use `_.groupBy` to build a dictionary mapping each `countyID` to an array of salaries, and we use our `countyValue` method to build an array of counties for our map.

We set `zoom` to `null` for now. This will also come into effect later.

In the `return` statement, we remove the "data loaded" indicator, and we add an `<svg>` element that's `1100` pixels wide and `500` pixels high. Inside, we put the `CountyMap` component with a bunch of properties. Some dataset stuff, some sizing and positioning stuff.

Step 2: CountyMap/index.js

We use `index.js` for one reason alone: to make imports and debugging easier. I learned this lesson the hard way so you don't have to.

CountyMap index.js

```
1 // src/components/CountyMap/index.js
2
3 export { default } from './CountyMap';
```

We export the default import from `./CountyMap.js`. That's it.

This allows us to import `CountyMap` from the directory without knowing about internal file structure. We *could* put all the code in this `index.js` file, but then stack traces are hard to read.

Putting a lot of code into `<directory>/index.js` files means that when you're looking at a stack trace, or opening different source files inside the browser, they're all going to be named `index.js`. Life is easier when components live inside a file named the same as the component you're using.

Step 3: CountyMap/CountyMap.js

Now here comes the fun part - declaratively drawing a map. You'll see why I love using React for dataviz.

We're using the full-feature integration and a lot of D3 magic for maps. I'm always surprised by how little code it takes to draw a map with D3.

Start with the imports: React, D3, lodash, topojson, the County component.

Import CountyMap dependencies

```
// src/components/CountyMap/CountyMap.js
import React, { Component } from 'react';
import * as d3 from 'd3';
import * as topojson from 'topojson';
import _ from 'lodash';

import County from './County';
```

Out of these, we haven't built `County` yet, and you haven't seen `topojson` before. It's a way of defining geographical data with JSON. We're going to use the `topojson` library to translate our geographical datasets into GeoJSON, which is another way of defining geo data with JSON.

I don't know why there are two, but TopoJSON produces smaller files, and GeoJSON can be fed directly into D3's geo functions. ¯\(ツ)/¯

Maybe it's a case of competing standards[41].

Constructor

Let's stub out the `CountyMap` component then fill it in with logic.

CountyMap stub

```
// src/components/CountyMap/CountyMap.js
class CountyMap extends Component {
    constructor(props) {
        super(props);

        this.updateD3(props);
    }

    componentWillReceiveProps(newProps) {
        this.updateD3(newProps);
    }

    updateD3(props) {

    }

    render() {
```

[41] https://xkcd.com/927/

```
        if (!this.props.usTopoJson) {
            return null;
        }else{
            return (

            );
        }
    }
}

export default CountyMap;
```

We'll set up default D3 state in `constructor` and keep it up to date in `updateD3`. To avoid repetition, we call `updateD3` in the constructor as well.

We need three D3 objects to build a choropleth map: a geographical projection, a path generator, and a quantize scale for colors.

D3 objects for a map

```
// src/components/CountyMap/CountyMap.js
class CountyMap extends Component {
    constructor(props) {
        super(props);

        this.projection = d3.geoAlbersUsa()
                            .scale(1280);
        this.geoPath = d3.geoPath()
                         .projection(this.projection);
        this.quantize = d3.scaleQuantize()
                          .range(d3.range(9));

        this.updateD3(props);
    }
```

You might remember geographical projections from high school geography. They map a sphere to a flat surface. We use `geoAlbersUsa` because it's made specifically to draw maps of the USA.

You can see the other projections D3 offers on the d3-geo Github page[42].

The `geoPath` generator takes a projection and returns a function that generates the `d` attribute of `<path>` elements. This is the most general way to specify SVG shapes. I won't go into explaining the `d` here, but it's an entire DSL[43].

[42] https://github.com/d3/d3-geo#projections

[43] https://developer.mozilla.org/en-US/docs/Web/SVG/Attribute/d

`quantize` is a D3 scale. We've dealt with the basics of scales in the D3 Axis example. This one splits a domain into 9 quantiles and assigns them specific values from the `range`.

Let's say our domain goes from 0 to 90. Calling the scale with any number between 0 and 9 would return 1. 10 to 19 returns 2 and so on. We'll use it to pick colors from an array.

updateD3

Keeping all of this up-to-date is easy, but we'll make it harder by adding a zoom feature. It won't work until we implement the filtering, but hey, we'll already have it by then! :D

CountyMap updateD3

```
1    // src/components/CountyMap/CountyMap.js
2    updateD3(props) {
3        this.projection
4            .translate([props.width / 2, props.height / 2])
5            .scale(props.width*1.3);
6
7        if (props.zoom && props.usTopoJson) {
8            const us = props.usTopoJson,
9                  statePaths = topojson.feature(us, us.objects.states).features,
10                 id = _.find(props.USstateNames, {code: props.zoom}).id;
11
12           this.projection.scale(props.width*4.5);
13
14           const centroid = this.geoPath.centroid(_.find(statePaths, {id: id})),
15                 translate = this.projection.translate();
16
17           this.projection.translate([
18               translate[0] - centroid[0] + props.width / 2,
19               translate[1] - centroid[1] + props.height / 2
20           ]);
21       }
22
23       if (props.values) {
24           this.quantize.domain([
25               d3.quantile(props.values, 0.15, d => d.value),
26               d3.quantile(props.values, 0.85, d => d.value)
27           ]);
28       }
29   }
```

There's a lot going on here.

The first part is okay. It `translates` (moves) the projection to the center of our drawing area and sets a scale property. The value was discovered experimentally and is different for every projection.

Then we do some weird stuff if `zoom` is defined.

We get the list of all US state features in our geo data, find the one we're `zoom`-ing on, and use the `geoPath.centroid` method to calculate its center point. This gives us a new coordinate to `translate` our projection onto.

The calculation in `.translate()` helps us align the center point of our `zoom` US state with the center of the drawing area.

While all of this is going on, we also tweak the `.scale` property to make the map bigger. This creates a zooming effect.

At the end of the `updateD3` function, we update the quantize scale's domain with new values. Using `d3.quantile` lets us offset the scale to produce a more interesting map. The values were discovered experimentally - they cut off the top and bottom of the range because there isn't much there. This brings higher contrast to the richer middle of the range.

render

After all of that work, the `render` method is a breeze. We prep the data then loop through it and render `County` elements.

CountyMap render

```
    // src/components/CountyMap/CountyMap.js
    render() {
        if (!this.props.usTopoJson) {
            return null;
        }else{
            const us = this.props.usTopoJson,
                  statesMesh = topojson.mesh(us, us.objects.states,
                                             (a, b) => a !== b),
                  counties = topojson.feature(us, us.objects.counties).features;

            const countyValueMap = _.fromPairs(
                this.props.values
                    .map(d => [d.countyID, d.value])
            );

            return (
                <g transform={`translate(${this.props.x}, ${this.props.y})`}>
```

```
18                      {counties.map((feature) => (
19                          <County geoPath={this.geoPath}
20                                  feature={feature}
21                                  zoom={this.props.zoom}
22                                  key={feature.id}
23                                  quantize={this.quantize}
24                                  value={countyValueMap[feature.id]} />
25                       ))}
26
27                          <path d={this.geoPath(statesMesh)}
28                                style={{fill: 'none',
29                                        stroke: '#fff',
30                                        strokeLinejoin: 'round'}} />
31                  </g>
32              );
33          }
34      }
```

We use the topojson library to grab data out of the `usTopoJson` dataset. `.mesh` calculates a mesh for US states - a thin line around the edges. `.feature` calculates the features for each county - fill in with color.

Mesh and feature aren't tied to states or counties by the way. It's just a matter of what you get back: borders or flat areas. What you need depends on what you're building.

We use `_.fromPairs` to build a dictionary that maps county identifiers to their values. Building it beforehand makes our code faster. You can read some details about the speed optimizations here[44].

As promised, all we have to do in the `return` statement is loop through the list of `counties` and render `County` components. Each gets a bunch of attributes and will return a `<path>` element that looks like a specific county.

For the US state borders, we use a single `<path>` element and use `this.geoPath` to generate the `d` property.

Step 4: CountyMap/County.js

The `County` component itself is built out of two parts: imports and color constants, and a component that returns a `<path>`. We did all the hard calculation work in `CountyMap`.

[44] https://swizec.com/blog/optimizing-react-choropleth-map-rendering/swizec/7302

Imports and color constants

```
// src/components/CountyMap/County.js
import React, { Component } from 'react';
import _ from 'lodash';

const ChoroplethColors = _.reverse([
    'rgb(247,251,255)',
    'rgb(222,235,247)',
    'rgb(198,219,239)',
    'rgb(158,202,225)',
    'rgb(107,174,214)',
    'rgb(66,146,198)',
    'rgb(33,113,181)',
    'rgb(8,81,156)',
    'rgb(8,48,107)'
]);

const BlankColor = 'rgb(240,240,240)'
```

We import React and lodash, then define some color constants. I got the `ChoroplethColors` from some example online, and `BlankColor` is a pleasant gray.

Now we need the `County` component itself.

County component

```
// src/components/CountyMap/County.js
class County extends Component {
    shouldComponentUpdate(nextProps, nextState) {
        const { zoom, value } = this.props;

        return zoom !== nextProps.zoom
            || value !== nextProps.value;
    }

    render() {
        const { value, geoPath, feature, quantize } = this.props;

        let color = BlankColor;

        if (value) {
            color = ChoroplethColors[quantize(value)];
```

```
        }

        return (
            <path d={geoPath(feature)}
                  style={{fill: color}}
                  title={feature.id} />
        );
    }
}

export default County;
```

The `render` method uses the `quantize` scale to pick the right color and returns a `<path>` element. `geoPath` generates the `d` attribute, we set style to `fill` the color, and we give our path a `title`.

`shouldComponentUpdate` is more interesting. It's a React lifecycle method that lets us specify which prop changes are relevant to our component re-rendering.

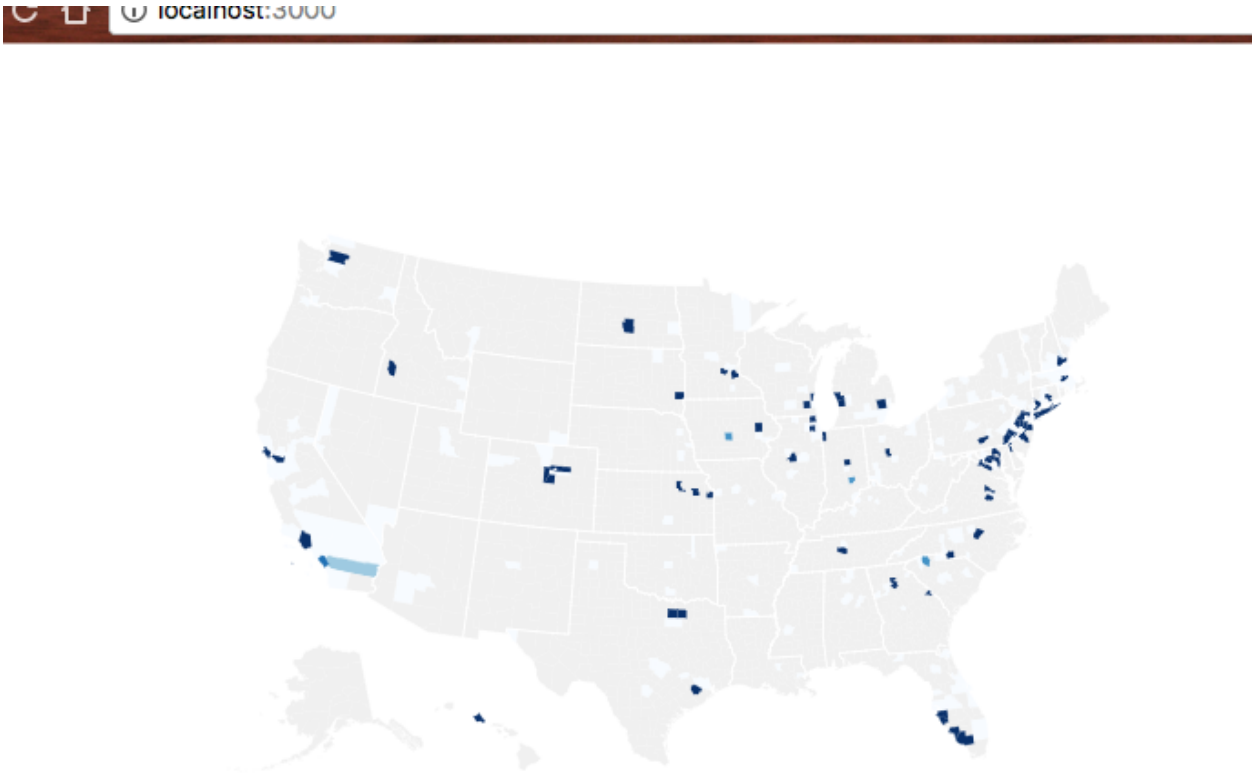
`CountyMap` passes complex props - `quantize`, `geoPath`, and `feature` - which are pass-by-reference instead of pass-by-value. That means React can't see when they produce different values, just when they are different instances.

This can lead to all 3,220 counties re-rendering every time a user does anything. But they only have to re-render if we change the map zoom or if the county gets a new value.

Using `shouldComponentUpdate` like this we can go from 3,220 DOM updates to the order of a few hundred. Big improvement in speed.

—

And with that, your browser should show you a map.



Choropleth map with shortened dataset

Turns out tech job visas just aren't that well distributed geographically. Most counties come out grey even with the full dataset.

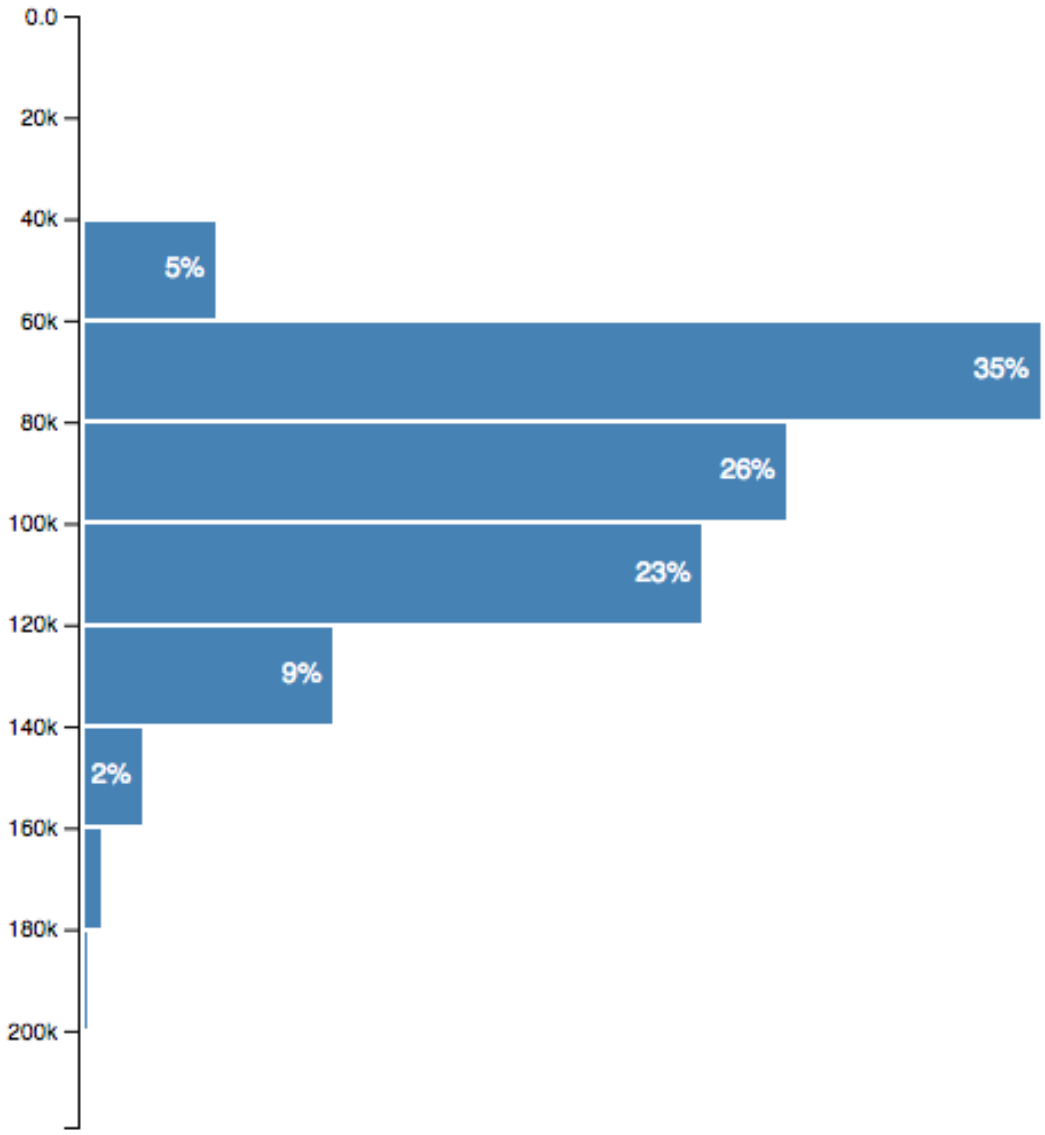
If that didn't work, consult this diff on Github[45].

[45] https://github.com/Swizec/react-d3js-step-by-step/commit/f4c1535e9c9ca4982c8f3c74cff9f739eb08c0f7

Render a Histogram of salaries

Knowing the median salary is great and all, but it doesn't tell you much about what you can expect. You need to know the distribution to see if it's more likely you'll get 140k or 70k.

That's what histograms are for. Give them a bunch of data, and they show its distribution. We're going to build one like this:



Basic histogram

In the shortened dataset, 35% of tech salaries fall between $60k and $80k, 26% between $80k and $100k etc. Throwing a random dice using this as your random distribution[46], you're far more likely to get 60k-80k than 120k-140k. Turns out this is a great way to gauge situations.

It's where fun statistics like "More people die from vending machines than shark attacks" come from. Which are you afraid of, vending machines or sharks? Stats say your answer should be heart disease[47]. ;)

Anyway, let's build a histogram. We'll start with changes in `App.js`, make a `Histogram` component using the full-feature approach, then add an `Axis` using the blackbox HOC approach. We're also going to add some CSS, finally.

Step 1: Prep App.js

You know the drill, don't you? Import some stuff, add it to the `render()` method in our `App` component.

Histogram imports

```
// src/App.js
import _ from 'lodash';

import './App.css';

import Preloader from './components/Preloader';
import { loadAllData } from './DataHandling';

import CountyMap from './components/CountyMap';
import Histogram from './components/Histogram';
```

We import `App.css` and the `Histogram` component. That's what I love about using Webpack - you can import CSS in JavaScript. We got the setup with `create-react-app`.

There are different schools of thought about how CSS should be used. Some say each component should have its own CSS files and that that's the whole reason we want JS-based imports anyway. Others think we shouldn't use CSS at all and should do styling in JavaScript.

Personally, I don't know. I like the idea of components coming with their own styling, but I find that makes them less reusable. Apps often want to specify their own styling.

Maybe a combination of default per-component styling and app-level overrides? Depends on your use case, I guess.

With the imports done, we can add `Histogram` to `App`'s render method.

[46] https://en.wikipedia.org/wiki/Probability_distribution
[47] https://www.cdc.gov/nchs/fastats/deaths.htm

Render Histogram in App

```
// src/App.js
// ...
render() {
    // ...
    return (
        <div className="App container">
            <h1>Loaded {this.state.techSalaries.length} salaries</h1>
            <svg width="1100" height="500">
                <CountyMap usTopoJson={this.state.usTopoJson}
                           USstateNames={this.state.USstateNames}
                           values={countyValues}
                           x={0}
                           y={0}
                           width={500}
                           height={500}
                           zoom={zoom} />
                <Histogram bins={10}
                           width={500}
                           height={500}
                           x="500"
                           y="10"
                           data={filteredSalaries}
                           axisMargin={83}
                           bottomMargin={5}
                           value={d => d.base_salary} />
            </svg>
        </div>
    );
}
```

We render the `Histogram` component with a bunch of props. They specify the dimensions we want, positioning, and pass data to the component. We're using `filteredSalaries` even though we haven't set up the filtering yet. One less line of code to change later :smile:

That's it. `App` is ready to render our `Histogram`.

Your browser should now show an error complaining about missing files.

Step 2: CSS changes

As mentioned, opinions vary on the best way to do styling in React apps. Some say stylesheets per component, some say styling inside JavaScript, others swear by global app styling.

The truth is somewhere in between. Do what best fits your project and team. We're going to stick to global stylesheets because it's the simplest.

Start by emptying out `src/App.css`. All that came with `create-react-app` must go. We don't need it.

Then add these 29 lines:

App.css stylesheet

```
.histogram .bar rect {
    fill: steelblue;
    shape-rendering: crispEdges;
}

.histogram .bar text {
    fill: #fff;
    font: 12px sans-serif;
}

button {
    margin-right: .5em;
    margin-bottom: .3em !important;
}

.row {
    margin-top: 1em;
}

.mean text {
    font: 11px sans-serif;
    fill: grey;
}

.mean path {
    stroke-dasharray: 3;
    stroke: grey;
    stroke-width: 1px;
}
```

We won't go into details about CSS here. Many better books have been written about it.

Generally speaking, we're making `.histogram` rectangles – the bars – blue, and labels white `12px` font. `buttons` and `.rows` have some spacing. This is for the user controls we'll add. And the `.mean` line is a dotted grey with grey `11px` text.

Yes, this is more CSS than we need for just the histogram. We're already here, so we might as well add it.

Adding our CSS before building the Histogram means it's going to look beautiful the first time around.

Step 3: Histogram component

We're following the full-feature integration approach for our Histogram component. React talks to the DOM, D3 calculates the props.

We'll use two components:

1. 'Histogram' handles the general layout, dealing with D3, and translating raw data into a histogram

2. `HistogramBar` draws a single bar and labels it

Let's start with the basics: a `Histogram` directory and an `index.js` file. It makes importing easier while keeping our code organized. I like to use dirs for components made out of multiple files.

Histogram index.js

```
// src/components/Histogram/index.js
import Histogram from './Histogram'

export default Histogram;
```

Import `Histogram` from `./Histogram` and export it as the `default` export. You could do it with a re-export: `export { default } from './Histogram'`. I'm not sure why I picked the long way. It's not *that* much more readable.

Great, now we need the `Histogram.js` file. We start with some imports, a default export, and a stubbed out `Histogram` class.

Histogram component stub

```
// src/components/Histogram/Histogram.js
import React, { Component } from 'react';
import * as d3 from 'd3';

class Histogram extends Component {
    constructor(props) {
        super();

        this.updateD3(props);
    }

    componentWillReceiveProps(newProps) {
        this.updateD3(newProps);
    }

    updateD3(props) {

    }

    makeBar(bar) {

    }

    render() {

    }
}

export default Histogram;
```

We need React and D3, and we set up `Histogram`. The `constructor` calls React's base constructor using `super()` and defers to `updateD3` to init default D3 properties. `componentWillReceiveProps` defers to `updateD3` to ensure D3 state stays in sync with React, and we'll use `makeBar` and `render` to render the SVG.

A note about D3 imports: D3v4 is split into multiple packages. We're using a * import here to get everything because that's easier to use. You should import specific packages when possible. It leads to smaller compiled code sizes and makes it easier for you and others to see what each file is using.

constructor

Now we should add D3 object initialization to the `constructor`. We need a D3 histogram and two scales: one for chart width and one for vertical positioning.

D3 initialization in Histogram constructor

```
// src/components/Histogram/Histogram.js
class Histogram extends Component {
    constructor(props) {
        super();

        this.histogram = d3.histogram();
        this.widthScale = d3.scaleLinear();
        this.yScale = d3.scaleLinear();

        this.updateD3(props);
    }
}
```

We've talked about scales before. Put in a number, get out a number. In this case, we're using linear scales for sizing and positioning.

`d3.histogram` is new in D3v4. It's a generator that takes a dataset and returns a histogram-shaped dataset. It's an array of arrays where the top level are bins and meta data and the children are "values in this bin".

You might know it as `d3.layout.histogram` from D3v3. I think the updated API is easier to use. You'll see what I mean in the `updateD3` method.

updateD3

updateD3 method in Histogram

```
// src/components/Histogram/Histogram.js
class Histogram extends Component {
    // ...
    updateD3(props) {
        this.histogram
            .thresholds(props.bins)
            .value(props.value);

        const bars = this.histogram(props.data),
              counts = bars.map((d) => d.length);
```

```
11
12          this.widthScale
13              .domain([d3.min(counts), d3.max(counts)])
14              .range([0, props.width-props.axisMargin]);
15
16          this.yScale
17              .domain([0, d3.max(bars, (d) => d.x1)])
18              .range([0, props.height-props.y-props.bottomMargin]);
19      }
20  }
```

First, we configure the `histogram` generator. We use `thresholds` to specify how many bins we want and `value` to specify a value accessor function. We get both from props passed into the `Histogram` component.

In our case, that's 20 bins, and the value accessor returns each data point's `base_salary`.

Then we call `this.histogram` on our dataset and use a `.map` to get an array of bins and count how many values went in each. We need them to configure our scales.

If you print the result of `this.histogram()`, you'll see an array structure where each entry holds metadata about the bin and the values it contains.

console.log(this.histogram())

We use this data to set up our scales.

`widthScale` has a range from the smallest (`d3.min`) bin to the largest (`d3.max`), and a range of `0` to width less a margin. We'll use it to calculate bar sizes.

`yScale` has a range from `0` to the largest `x1` coordinate we can find in a bin. Bins go from `x0` to `x1`, which reflects the fact that most histograms are horizontally oriented. Ours is vertical so that our labels are easier to read. The range goes from `0` to the maximum height less a margin.

Now let's render this puppy.

render

Histogram.render

```
// src/components/Histogram/Histogram.js
class Histogram extends Component {
    // ...
    render() {
        const translate = `translate(${this.props.x}, ${this.props.y})`,
              bars = this.histogram(this.props.data);

        return (
            <g className="histogram" transform={translate}>
                <g className="bars">
                    {bars.map(this.makeBar.bind(this))}
                </g>
            </g>
        );
    }
```

We set up a `translate` SVG transform and run our histogram generator. Yes, that means we're running it twice for every update, once in `updateD3` and once in `render`.

I tested making it more efficient, and I didn't see much improvement in overall performance. It did make the code more complex, though.

Our render method returns a `<g>` grouping element transformed to the position given in props and walks through the `bars` array, calling `makeBar` for each. Later, we're going to add an `Axis` as well.

This is a great example of React's declarativeness. We have a bunch of stuff, and all it takes to render is a loop. No worrying about how it renders, where it goes, or anything like that. Walk through data, render, done.

makeBar

`makeBar` is a function that takes a histogram bar's metadata and returns a `HistogramBar` component. We use it to make the declarative loop more readable.

Histogram.makeBar

```
// src/components/Histogram/Histogram.js
class Histogram extends Component {
    // ...
    makeBar(bar) {
        let percent = bar.length/this.props.data.length*100;

        let props = {percent: percent,
                     x: this.props.axisMargin,
                     y: this.yScale(bar.x0),
                     width: this.widthScale(bar.length),
                     height: this.yScale(bar.x1 - bar.x0),
                     key: "histogram-bar-"+bar.x0}

        return (
            <HistogramBar {...props} />
        );
    }
```

See, we're calculating `props` and feeding them into `HistogramBar`. Putting it in a separate function just makes the `.map` construct in `render` easier to read. There's a lot of props to calculate.

Some, like `axisMargin` we pass through, others like `width` and `height` we calculate using our scales.

Setting the `key` prop is important. React uses it to tell the bars apart and only re-render those that change.

Step 4: HistogramBar (sub)component

Before we can see the histogram, we need another component: `HistogramBar`. We *could* have shoved all of it in the `makeBar` function, but I think it makes sense to keep it separate.

I like to put subcomponents like this in the same file as the main component. You can put it in its own file, if you feel that's cleaner. You'll have to add an `import` if you do that.

HistogramBar component

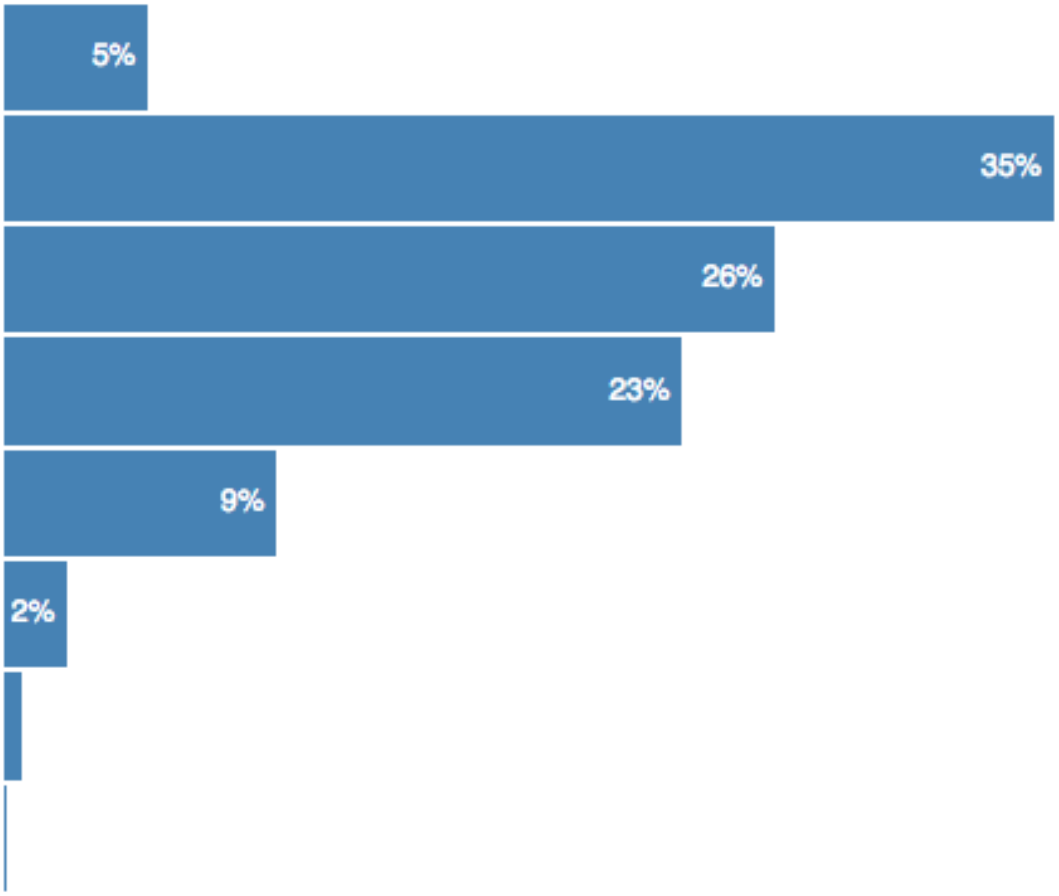
```
// src/components/Histogram/Histogram.js
const HistogramBar = ({ percent, x, y, width, height }) => {
    let translate = `translate(${x}, ${y})`,
        label = percent.toFixed(0)+'%';

    if (percent < 1) {
        label = percent.toFixed(2)+"%";
    }

    if (width < 20) {
        label = label.replace("%", "");
    }

    if (width < 10) {
        label = "";
    }

    return (
        <g transform={translate} className="bar">
            <rect width={width}
                  height={height-2}
                  transform="translate(0, 1)">
            </rect>
            <text textAnchor="end"
                  x={width-5}
                  y={height/2+3}>
                {label}
            </text>
        </g>
    );
}
```

As far as functional stateless components go, this one's pretty long. Most of it goes into deciding how much precision to render in the label, so it's okay.

We start with an SVG translate – you'll see this a few more times – and a default `label`. Then we update the label based on the bar size and its value.

When we have a label we like, we return a `<g>` grouping element with a rectangle and a text. They're both positioned based on the `width` and `height` of the bar.

You should now see a histogram.



Histogram without axis

Step 5: Axis HOC

Our histogram is pretty, but it needs an axis to be useful. You've already learned how to implement an axis when we talked about blackbox integration. We're going to use the same approach and copy those concepts into the real project.

D3blackbox

We start with the D3blackbox higher order component. Same as before, except we put it in `src/components`. Then again, I should probably just suck it up and make an npm package for it.

D3blackbox HOC

```
// src/components/D3blackbox.js
import React, { Component } from 'react';

export default function D3blackbox(D3render) {
  return class Blackbox extends Component {
    componentDidMount() { D3render.call(this); }
    componentDidUpdate() { D3render.call(this) }

    render() {
      const { x, y } = this.props;
      return <g transform={`translate(${x}, ${y})`} ref="anchor" />;
    }
  }
}
```

Take a `D3render` function, call it on `componentDidMount` and `componentDidUpdate` to keep things in sync, and render a positioned anchor element for `D3render` to hook into.

Axis component

With `D3blackbox`, we can reduce the `Axis` component to a wrapped function. We're implementing the `D3render` method.

Axis component using D3blackbox

```
// src/components/Histogram/Axis.js
import * as d3 from 'd3';
import D3blackbox from '../D3blackbox';

const Axis = D3blackbox(function () {
    const axis = d3.axisLeft()
                   .tickFormat(d => `${d3.format(".2s")(d)}`)
                   .scale(this.props.scale)
                   .ticks(this.props.data.length);

    d3.select(this.refs.anchor)
      .call(axis);
})

export default Axis;
```

We use D3's `axisLeft` generator, configure its `tickFormat`, give it a `scale` to use, and specify how many `ticks` we want. Then `select` the anchor element rendered by `D3blackbox` and `call` the axis generator on it.

Yes, this `Axis` works only for our specific use case. That's okay! No need to make things general when you're only using them once.

Add it to Histogram

To render our new `Axis`, we have to add it to the `Histogram` component. The process takes two steps:

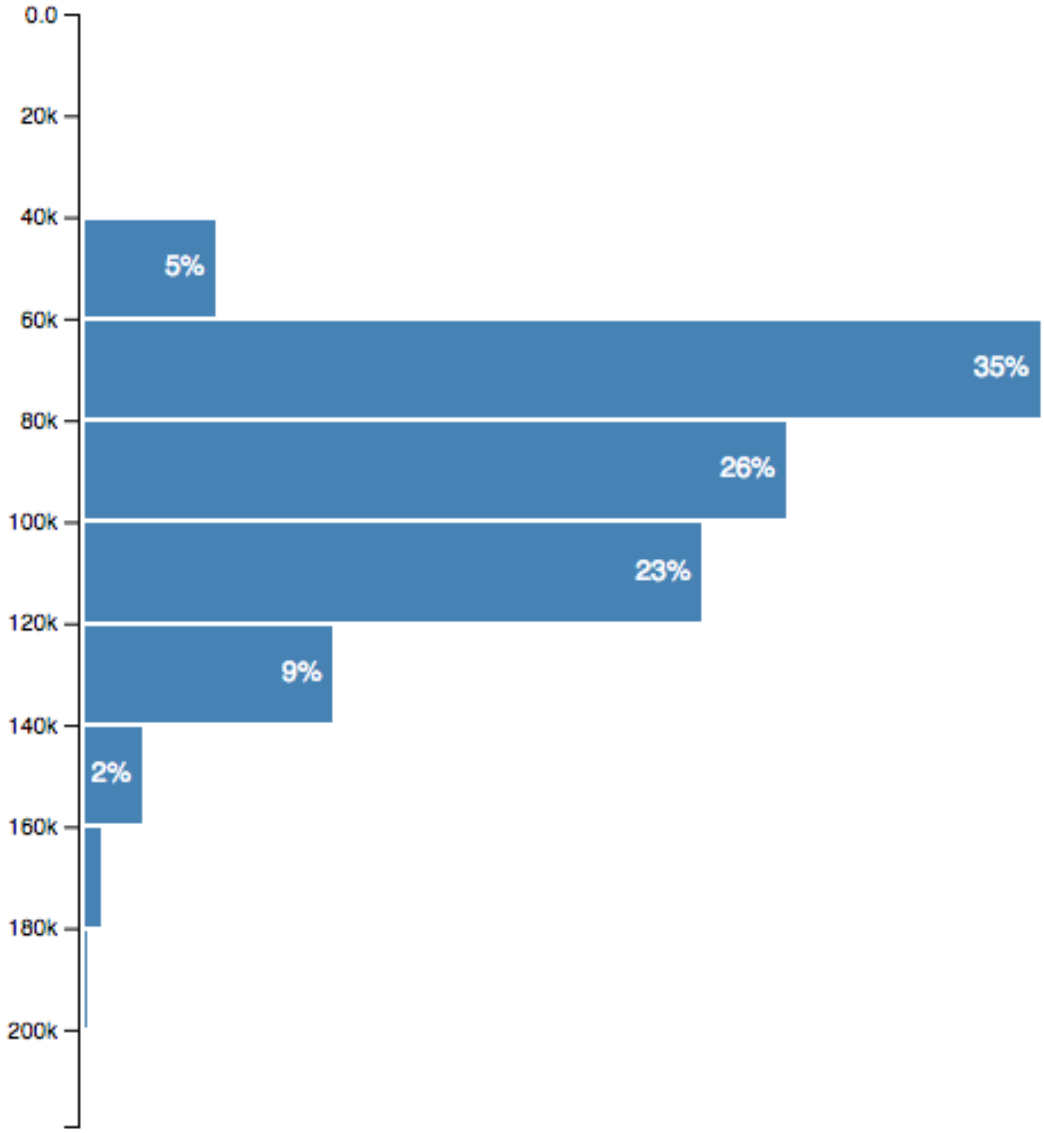
1. Import `Axis` component
2. Render it

Import and render Axis

```
// src/components/Histogram/Histogram.js
import React, { Component } from 'react';
import * as d3 from 'd3';

import Axis from './Axis';

// ...
class Histogram extends Component {
    // ...
    render() {
        const translate = `translate(${this.props.x}, ${this.props.y})`,
              bars = this.histogram(this.props.data);

        return (
            <g className="histogram" transform={translate}>
                <g className="bars">
                    {bars.map(this.makeBar.bind(this))}
                </g>
                <Axis x={this.props.axisMargin-3}
                      y={0}
                      data={bars}
                      scale={this.yScale} />
            </g>
        );
    }
```

We import our `Axis` component and add it to `Histogram`'s `render` method with some props. It takes an `x` and `y` coordinate, the `data`, and a `scale`.

An axis appears.



Basic histogram with axis

If that didn't work, try comparing your changes to this diff on Github[48].

[48] https://github.com/Swizec/react-d3js-step-by-step/commit/02a40899e348587a909e97e8f18ecf468e2fe218

Make it understandable - meta info

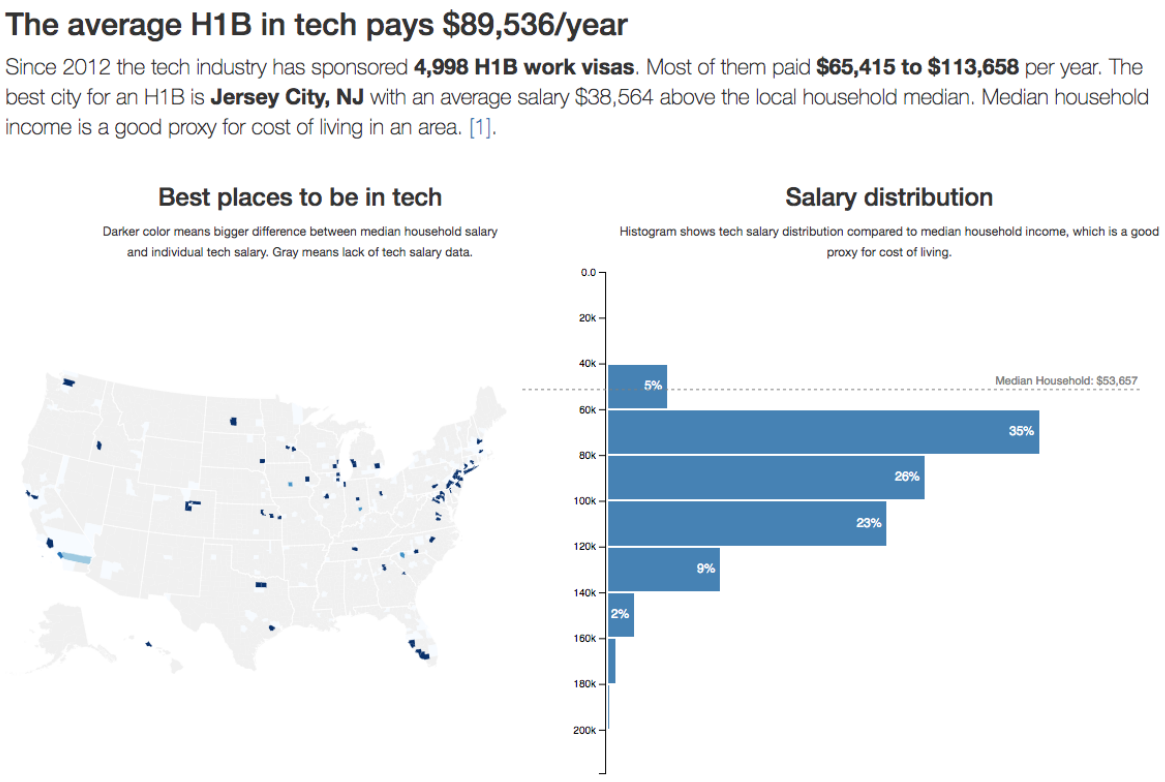
You've come so far! There's a US map and a histogram. They're blue and shiny and you look at them and you go *"Huh?"*.

The key to a good data visualization is telling users what it means. You can do that with a title and a description. Just tell them. The picture is there to give support to the words. The words are there to tell you what's in the picture.

Let's add those words.

We're adding a dynamic title and description, and a median line on the histogram. It's dynamic because we're adding user controls later, and we want the pictures and the words to stay in sync.

At the end of this section, you'll have a full visualization of the shortened dataset.



Full visualization without user controls

Dynamic title

We begin with the title because it shows up first.

We start with an import in `App.js` and add it to the render method. You know the drill :smile:

Adding Title to main App component

```
// src/App.js
import CountyMap from './components/CountyMap';
import Histogram from './components/Histogram';
import { Title } from './components/Meta';

class App extends Component {
    state = {
        techSalaries: [],
        countyNames: [],
        medianIncomes: [],
        filteredBy: {
            USstate: '*',
            year: '*',
            jobTitle: '*'
        }
    }

    // ...

    render() {
        // ..
        return (
            <div className="App container">
                <Title data={filteredSalaries}
                       filteredBy={this.state.filteredBy} />
                // ...
            </div>
        )
    }
```

Ok, I lied. We did a lot more than just imports and adding a render.

We also set up the `App` component for future user-controlled data filtering. The `filteredBy` key in `state` tells us what the user is filtering by. There 3 options: `USstate`, `year`, and `jobTitle`. We set them to "everything" by default.

We added them now so that we can immediately write our `Title` component in a filterable way. That means we don't have to make changes later.

As you can see, the props `Title` takes are `data` and `filteredBy`.

Prep Meta component

Before we begin the `Title` component, there are a few things to take care of. Our meta components work together for a common purpose – showing meta data. Grouping them in a directory makes sense.

So we make a `components/Meta` directory and add an `index.js`. It makes importing easier.

Meta index.js

```
1 // src/components/Meta/index.js
2 export { default as Title } from './Title';
```

You're right, using re-exports looks better than the roundabout way we used in `Histogram/index.js`. Lesson learned.

You need the `USStatesMap` file as well. It translates US state codes to full names. You should get it from Github[49] and save it as `components/Meta/USStatesMap.js`.

We'll use it when creating titles and descriptions.

Implement Title

We're building two types of titles based on user selection. If both `years` and `US state` were selected, we return `In {US state}, the average {job title} paid ${mean}/year in {year}`. If not, we return `{job title} paid ${mean}/year in {state} in {year}`.

I know, it's confusing. They look like the same sentence turned around. Notice the *and*. First option when *both* are selected, second when either/or.

We start with imports, a stub, and a default export.

[49] https://github.com/Swizec/react-d3js-step-by-step/blob/4f94fcd1c3caeb0fc410636243ca99764e27c5e6/src/components/Meta/USStatesMap.js

Title component stub

```
// src/components/Meta/Title.js
import React, { Component } from 'react';
import { scaleLinear } from 'd3-scale';
import { mean as d3mean, extent as d3extent } from 'd3-array';

import USStatesMap from './USStatesMap';

class Title extends Component {
    get yearsFragment() {
    }

    get USstateFragment() {
    }

    get jobTitleFragment() {
    }

    get format() {
    }

    render() {
    }
}

export default Title;
```

We import only what we need from D3's `d3-scale` and `d3-array` packages. I consider this best practice until you're importing so much that it gets messy to look at.

In the `Title` component, we have `4` getters and a render. Getters are ES6 functions that work like dynamic properties. You specify a function without arguments, and you use it without `()`. It's neat.

The getters

1. `yearsFragment` describes the selected year
2. `USstateFragment` describes the selected US state
3. `jobTitleFragment` describes the selected job title
4. `format` returns a number formatter

We can implement `yearsFragment`, `USstateFragment`, and `format` in one code sample. They're short.

3 short getters in Title

```
// src/components/Meta/Title.js
class Title extends Component {
    get yearsFragment() {
        const year = this.props.filteredBy.year;

        return year === '*' ? "" : `in ${year}`;
    }

    getteFragment() {
        const USstate = this.props.filteredBy.USstate;

        return USstate === '*' ? "" : USStatesMap[USstate.toUpperCase()];
    }

    // ...

    get format() {
        return scaleLinear()
                .domain(d3extent(this.props.data, d => d.base_salary))
                .tickFormat();
    }
```

In both `yearsFragment` and `USstateFragment`, we get the appropriate value from Title's `filteredBy` prop, then return a string with the value or an empty string.

We rely on D3's built-in number formatters to build `format`. Linear scales have the one that turns `10000` into `10,000`. Tick formatters don't work well without a `domain`, so we define it. We don't need a range because we never use the scale itself.

`format` returns a function, which makes it a higher order function[50]. Being a getter makes it really nice to use: `this.format()`. Looks just like a normal function call :D

The `jobTitleFragment` getter is conceptually no harder than `yearsFragment` and `USstateFragment`, but it comes with a few more conditionals.

[50] https://en.wikipedia.org/wiki/Higher-order_function

Title.jobTitleFragment

```
// src/components/Meta/Title.js
class Title extends Component {
    // ...
    get jobTitleFragment() {
        const { jobTitle, year } = this.props.filteredBy;
        let title = "";

        if (jobTitle === '*') {
            if (year === '*') {
                title = "The average H1B in tech pays";
            }else{
                title = "The average tech H1B paid";
            }
        }else{
            if (jobTitle === '*') {
                title = "H1Bs in tech pay";
            }else{
                title = `Software ${jobTitle}s on an H1B`;

                if (year === '*') {
                    title += " make";
                }else{
                    title += " made";
                }
            }
        }

        return title;
    }
    // ...
}
```

We're dealing with the `(jobTitle, year)` combination. Each influences the other when building the fragment for a total 4 different options.

The render

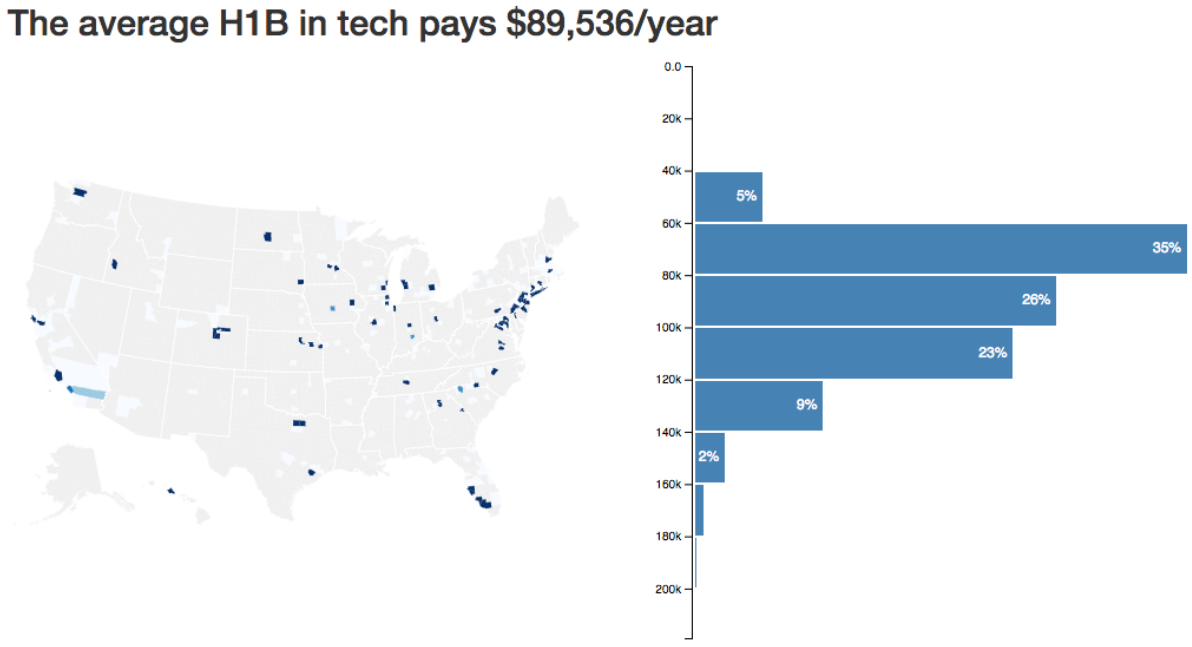
We put all this together in the `render` method. A conditional decides which of the two situations we're in, and we return an `<h2>` tag with the right text.

Title.render

```
// src/components/Meta/Title.js
class Title extends Component {
    // ...
    render() {
        const mean = this.format(d3mean(this.props.data, d => d.base_salary));

        let title;

        if (this.yearsFragment && this.USstateFragment) {
            title = (
                <h2>
                    In {this.USstateFragment}, {this.jobTitleFragment}
                    ${mean}/year {this.yearsFragment}
                </h2>
            );
        }else{
            title = (
                <h2>
                    {this.jobTitleFragment} ${mean}/year
                    {this.USstateFragment ? `in ${this.stateFragment}` : ''}
                    {this.yearsFragment}
                </h2>
            );
        }

        return title;
    }
}
```

Calculate the mean value using `d3.mean` with a value accessor, turn it into a pretty number with `this.format`, then use one of two string patterns to make a `title`.

And a title appears.



Dataviz with title

If it doesn't, consult this diff on Github[51].

Dynamic description

You know what? The dynamic description component is pretty much the same as the title. It's just longer and more complex and uses more code. It's interesting, but not super relevant to the topic of this book.

So rather than explain it all here, I'm going to give you a link to the diff on Github[52]

We use the same approach as before:

1. Add imports in `App.js`
2. Add component to `App` render
3. Add re-export to `components/Meta/index.js`
4. Implement component in `components/Meta/Description.js`
5. Use getters for sentence fragments
6. Play with conditionals to construct different sentences

142 lines of mundane code.

All the interesting complexity goes into finding the richest city and county. That part looks like this:

[51] https://github.com/Swizec/react-d3js-step-by-step/commit/4f94fcd1c3caeb0fc410636243ca99764e27c5e6
[52] https://github.com/Swizec/react-d3js-step-by-step/commit/032fe6e988b903b6d86a60d2f0404456785e180f

Richest county calculation

```
// src/components/Meta/Description.js
get countyFragment() {
        const byCounty = _.groupBy(this.props.data, 'countyID'),
              medians = this.props.medianIncomesByCounty;

        let ordered = _.sortBy(
            _.keys(byCounty)
             .map(county => byCounty[county])
             .filter(d => d.length/this.props.data.length > 0.01),
            items => d3mean(items,
                            d => d.base_salary) - medians[items[0].countyID][0].medianI\
ncome);

        let best = ordered[ordered.length-1],
            countyMedian = medians[best[0].countyID][0].medianIncome;

        // ...
}
```

We group the dataset by county, then sort counties by their income delta. We look only at counties that are bigger than 1% of the entire dataset. And we define income delta as the difference between a county's median household income and the median tech salary in our dataset.

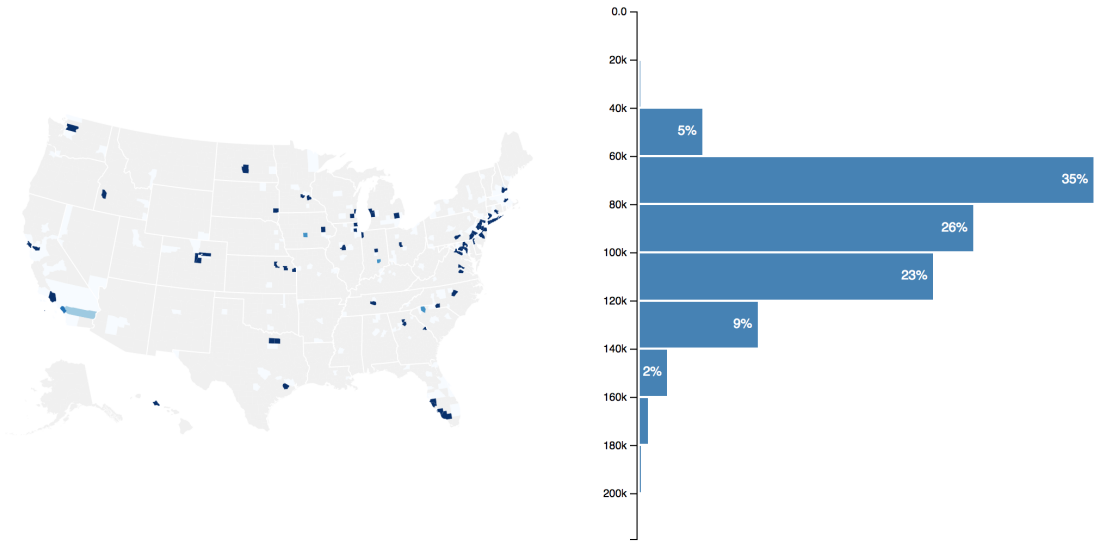
Now that I think about it, this is not very efficient. We should've just looked for the maximum value. That would've been faster, but hey, it works :smile:

We use basically the same process to get the best city.

Yes, you're right. These should both have been separate functions. Putting them in the `countyFragment` method smells funny.

If you follow along the description Github diff[53], or copy pasta, your visualization should now have a description.

[53] https://github.com/Swizec/react-d3js-step-by-step/commit/032fe6e988b903b6d86a60d2f0404456785e180f

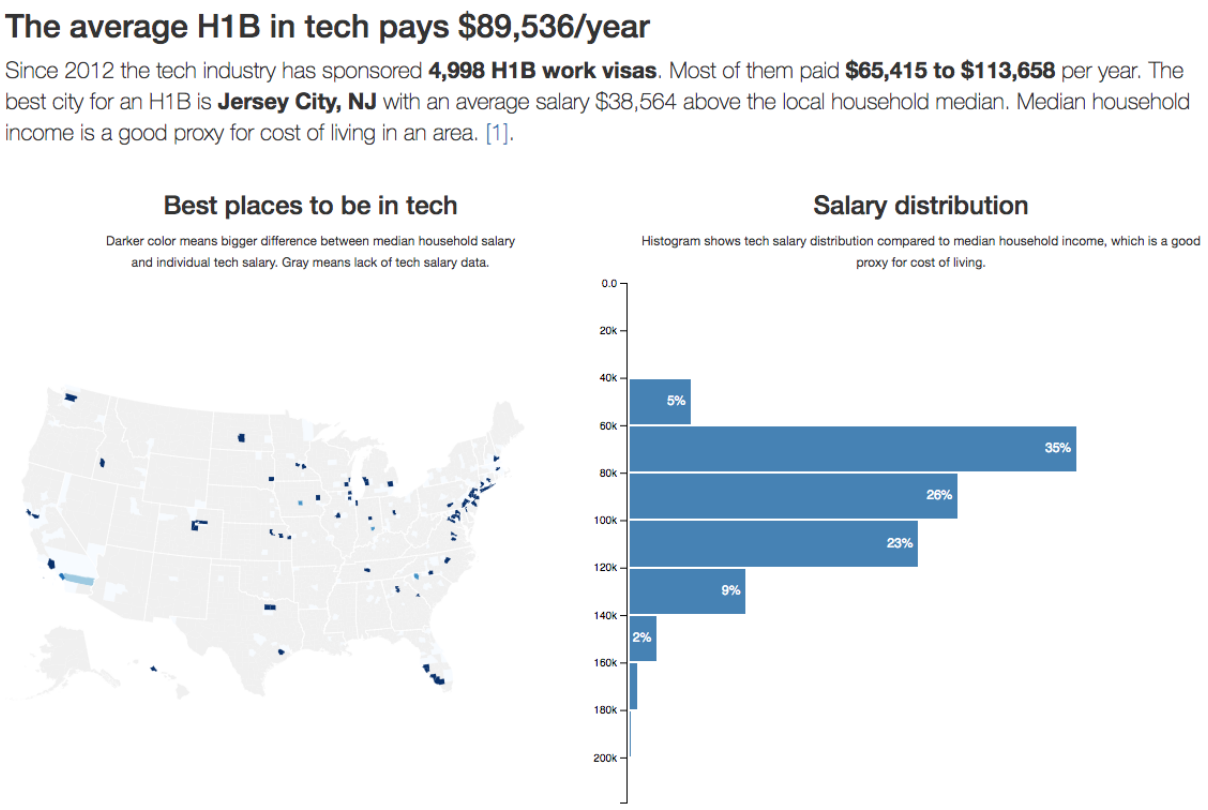


Dataviz with Title and Description

Another similar component is the `GraphDescription`. It shows a small description on top of each chart that explains how to read the picture. Less "Here's a key takeaway", more "color means X".

You can follow this diff on Github[54] to implement it. Same approach as `Title` and `Description`.

[54] https://github.com/Swizec/react-d3js-step-by-step/commit/37b5222546c3f8f58f3147ce0bef6a3c1afe1b47



Dataviz with all descriptions

Median household line

Now here's a more interesting component: the median dotted line. It gives us a direct comparison between the histogram's distribution and the median household income in an area. I'm not sure people understand it at a glance, but I think it's cool.

We're using the full-feature integration approach, and prepping `App.js` first, then implementing the component.

Step 1: App.js

Inside `src/App.js`, we first have to add an import, then extract the median household value from state, and in the end, add `MedianLine` to the render method.

Let's see if we can do it in a single code block :smile:

Adding MedianLine to App.js

```
// src/App.js
import Histogram from './components/Histogram';
import { Title, Description, GraphDescription } from './components/Meta';
import MedianLine from './components/MedianLine';

class App extends Component {
    // ...
    render() {
        // ...
        let zoom = null,
            medianHousehold = this.state.medianIncomesByUSState['US'][0]
                                    .medianIncome;

        return (
            // ...
            <svg width="1100" height="500">
                <CountyMap // ... />
                <Histogram // ... />
                <MedianLine data={filteredSalaries}
                            x={500}
                            y={10}
                            width={600}
                            height={500}
                            bottomMargin={5}
                            median={medianHousehold}
                            value={d => d.base_salary} />
            </svg>
        )
    }
}
```

You probably don't remember `medianIncomesByUSState` anymore. We set it up way back when tying datasets together. It groups our salary data by US state.

See, using good names helps :smile:

When rendering `MedianLine`, we give it sizing and positioning props, the dataset, a `value` accessor, and the median value to show. Yes, we can make it smart enough to calculate the median, but the added flexibility of a prop felt right.

Step 2: MedianLine

The `MedianLine` component looks a lot like what you're already used to. Some imports, a `constructor` that sets up D3 objects, an `updateD3` method that keeps them in sync, and a `render` method that outputs SVG.

MedianLine component stub

```
// src/components/MedianLine.js
import React, { Component } from 'react';
import * as d3 from 'd3';

class MedianLine extends Component {
    componentWillMount() {
        this.yScale = d3.scaleLinear();

        this.updateD3(this.props);
    }

    componentWillReceiveProps(newProps) {
        this.updateD3(newProps);
    }

    updateD3(props) {
        this.yScale
            .domain([0,
                     d3.max(props.data, props.value)])
            .range([0, props.height-props.y-props.bottomMargin]);
    }

    render() {

    }
}

export default MedianLine;
```

Standard stuff, right? You've seen it all before. Bear with me, please. I know you're great, but I gotta explain this for everyone else :smile:

We have the base wiring for a D3-enabled component, and we set up a linear scale that we'll use for vertical positioning. The scale has a `domain` from `0` to `max` value in dataset and a range from `0` to height less margins.

MedianLine render

```
// src/components/MedianLine.js
class MedianLine extends Component {
    // ...
    render() {
        const median = this.props.median || d3.median(this.props.data,
                                                       this.props.value),
              line = d3.line()([[0, 5],
                                [this.props.width, 5]]),
              tickFormat = this.yScale.tickFormat();

        const translate = `translate(${this.props.x}, ${this.yScale(median)})`,
              medianLabel = `Median Household: $${tickFormat(median)}`;

        return (
            <g className="mean" transform={translate}>
                <text x={this.props.width-5} y="0" textAnchor="end">
                    {medianLabel}
                </text>
                <path d={line}></path>
            </g>
        );
```

We use the median value from props, or calculate our own, if needed. Just like I promised.

We also set up a `translate` SVG transform and the `medianLabel`. The return statement builds a `<g>` grouping element, transformed to our desired position, containing a `<text>` for our label, and a `<path>` for the line.

But how we get the `d` attribute for the path, that's interesting. We use a `line` generator from D3.

Line generator

```
line = d3.line()([[0, 5],
                  [this.props.width, 5]]);
```

It comes from the d3-shape[55] package and generates splines, or polylines. By default, it takes an array of points and builds a line through all of them. A line from `[0, 5]` to `[width, 5]` in our case.

That makes it span the entire width and leaves 5px of room for the label. We're using a `transform` on the entire group to vertically position the final element.

[55] https://github.com/d3/d3-shape#lines

We're using `d3.line` in the most basic way possible, but it's really flexible. You can even build curves.

Remember, we styled the `medianLine` when we did histogram styles earlier.

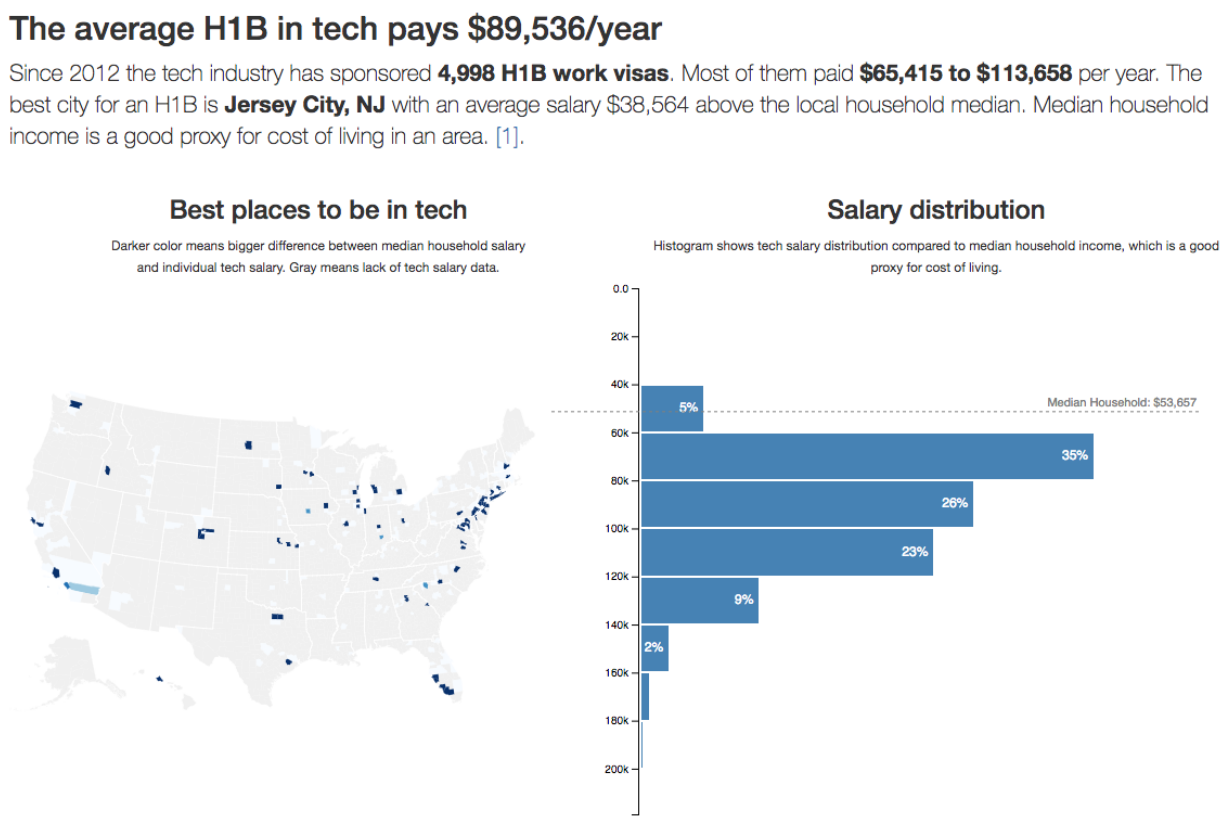
Histogram css

```
.mean text {
    font: 11px sans-serif;
    fill: grey;
}

.mean path {
    stroke-dasharray: 3;
    stroke: grey;
    stroke-width: 1px;
}
```

The `stroke-dasharray` is what makes it dashed. `3` means each `3px` dash is followed by a `3px` blank. You can use any pattern you like[56].

You should now see a median household salary line overlaid on your histogram.

[56] https://developer.mozilla.org/en-US/docs/Web/SVG/Attribute/stroke-dasharray



Median line over histogram

Yep, almost everyone in tech makes more than the median household. Crazy, huh? I think it is.

If that didn't work, consult the diff on Github[57].

[57] https://github.com/Swizec/react-d3js-step-by-step/commit/1fd055e461184fb8dc8dd509edb3a6a683c995fe

Add user controls for data slicing and dicing

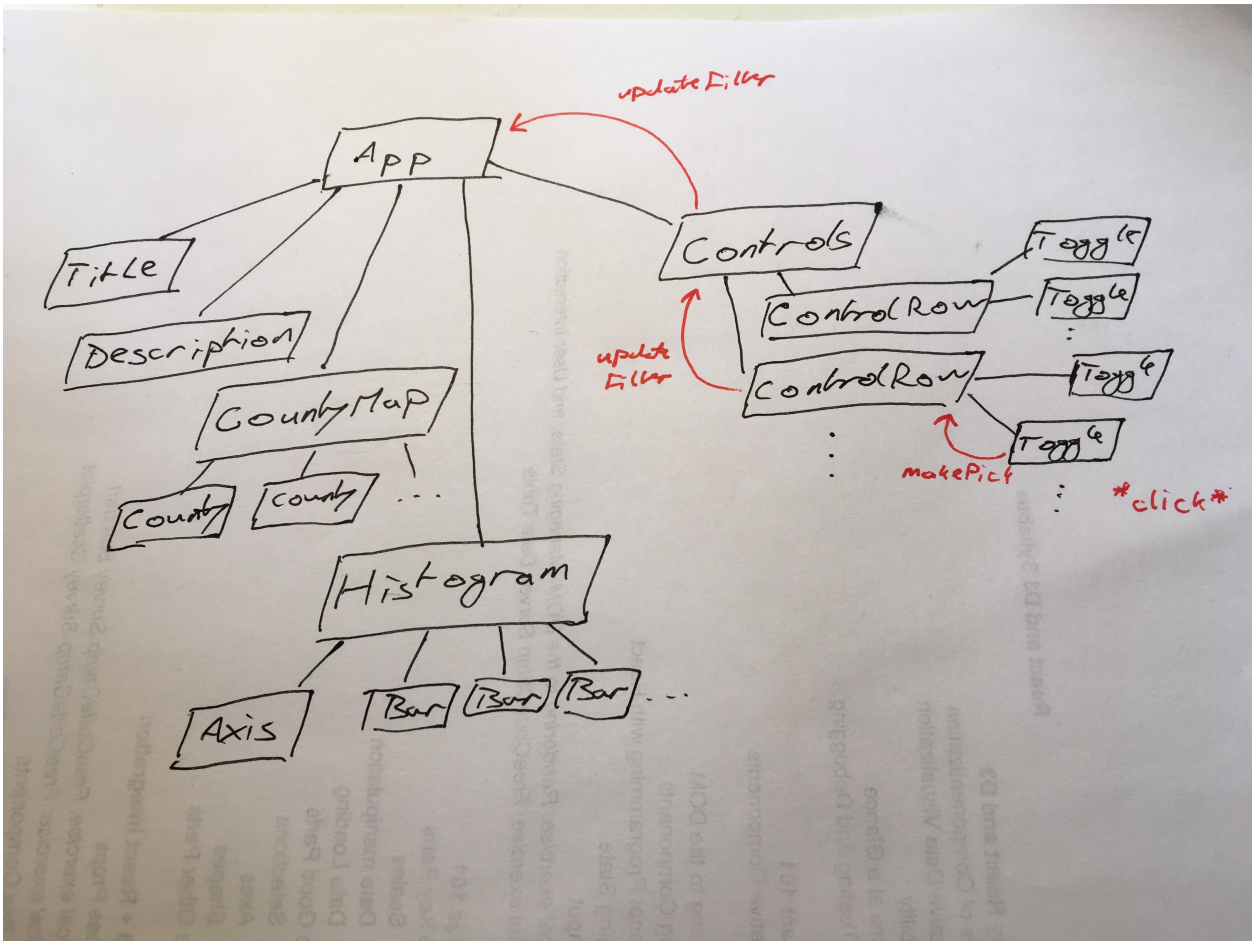
Now comes the fun part. All that extra effort we put into making our components aware of filtering, and it all comes down to this: User controls.

Here's what we're building:

User controlled filters

It's a set of filters for users to slice and dice our visualization. The shortened dataset gives you 2 years, 12 job titles, and 50 US states. You'll get 5 years and many more job titles with the full dataset.

We're using the architecture we discussed earlier to make it work. Clicking buttons updates a filter function and communicates it all the way up to the `App` component. `App` then uses it to update `this.state.filteredSalaries`, which triggers a re-render and updates our dataviz.



Architecture sketch

We're building in 4 steps, top to bottom:

1. Update `App.js` with filtering and a `<Controls>` render
2. Build a `Controls` component, which builds the filter based on inputs
3. Build a `ControlRow` component, which handles a row of buttons
4. Build a `Toggle` component, which is a button

We'll go through the files linearly. That makes them easier for me to explain and easier for you to understand, but that also means there's going to be a long period where all you're seeing is an error like this:

```
Failed to compile.

Error in ./src/components/Controls/index.js

/Users/Swizec/Documents/random-coding/react-d3js-step-by-step/src/components/Controls/index.js
  82:18  error  'ControlRow' is not defined  react/jsx-no-undef
  87:18  error  'ControlRow' is not defined  react/jsx-no-undef
  92:18  error  'ControlRow' is not defined  react/jsx-no-undef

✖ 3 problems (3 errors, 0 warnings)
```

Controls error during coding

If you want to see what's up during this process, just remove an import or two and maybe a thing from render. For instance, it's complaining about `ControlRow` in this screenshot. Remove the `ControlRow` import on top and delete `<ControlRow ... />` from render. The error goes away, and you see what you're doing.

Step 1: Update App.js

All right, you know the drill. Add imports, tweak some things, add to render. We have to import `Controls`, set up filtering, update the map's `zoom` prop, and render a white rectangle and `Controls`.

The white rectangle makes it so the zoomed-in map doesn't cover up the histogram. I'll explain when we get there.

Imports and filter updates in App.js

```
// src/App.js
import MedianLine from './components/MedianLine';

import Controls from './components/Controls';

class App extends Component {
    state = {
        // ...
        medianIncomes: [],
        salariesFilter: () => true,
        filteredBy: {
            // ...
        }
    }

    // ...

    updateDataFilter(filter, filteredBy) {
```

```
        this.setState({
            salariesFilter: filter,
            filteredBy: filteredBy
        });
    }

    render() {
        // ...
    }
}
```

We import the `Controls` component and add a default `salariesFilter` function to `this.state`. The `updateDataFilter` method passes the filter function and `filteredBy` dictionary from arguments to App state. We'll use it as a callback in `Controls`.

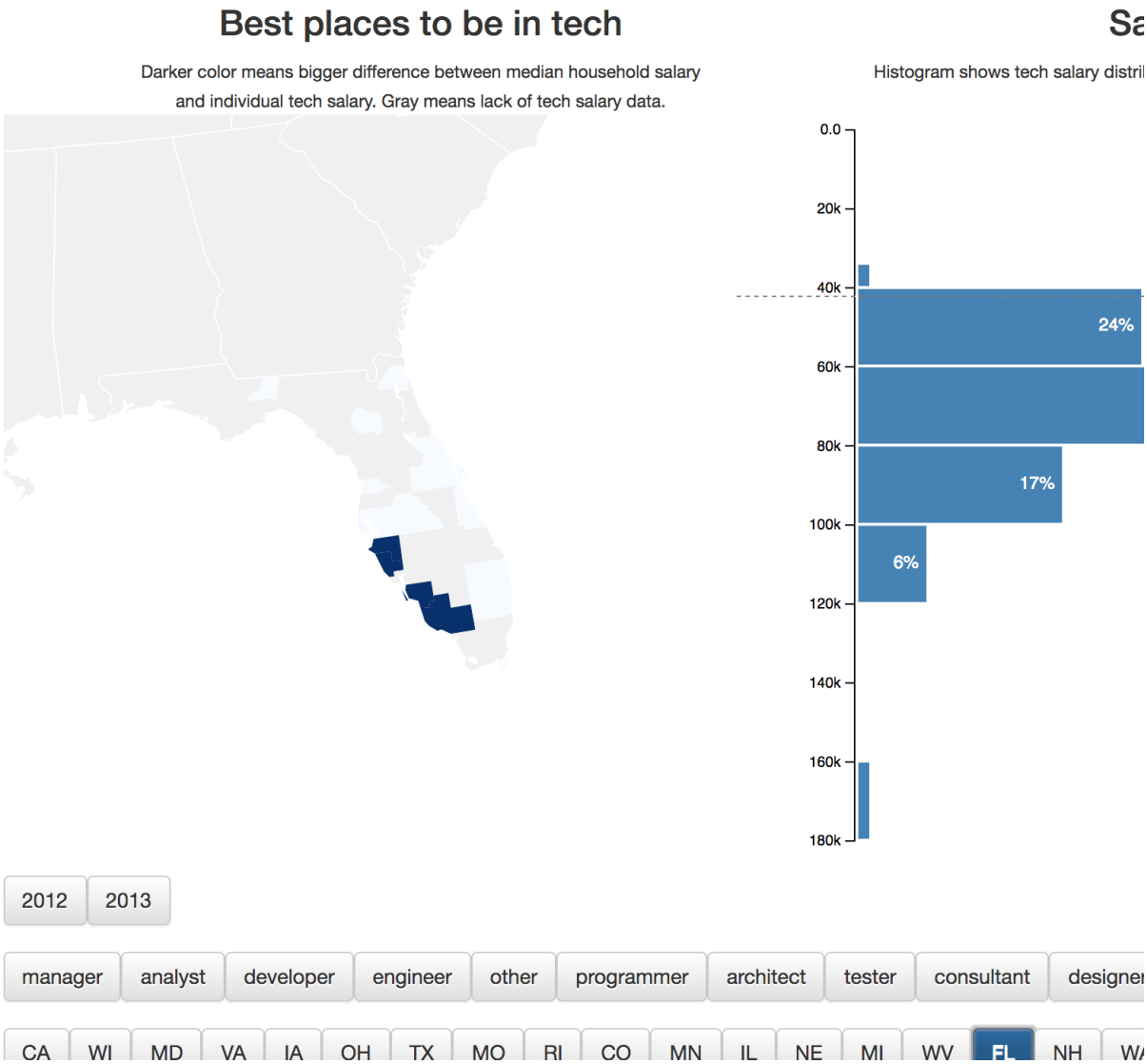
The rest of filtering setup happens in the render method.

Filtering data and updating map zoom in App render

```
// src/App.js
class App extends Component {
    // ...

    render() {
        // ...
        const filteredSalaries = this.state.techSalaries
        const filteredSalaries = this.state.techSalaries
                                    .filter(this.state.salariesFilter)

        // ...

        let zoom = null,
            medianHousehold = // ...
        if (this.state.filteredBy.USstate !== '*') {
            zoom = this.state.filteredBy.USstate;
            medianHousehold = d3.mean(this.state.medianIncomesByUSState[zoom],
                                      d => d.medianIncome);
        }

        // ...
    }
}
```

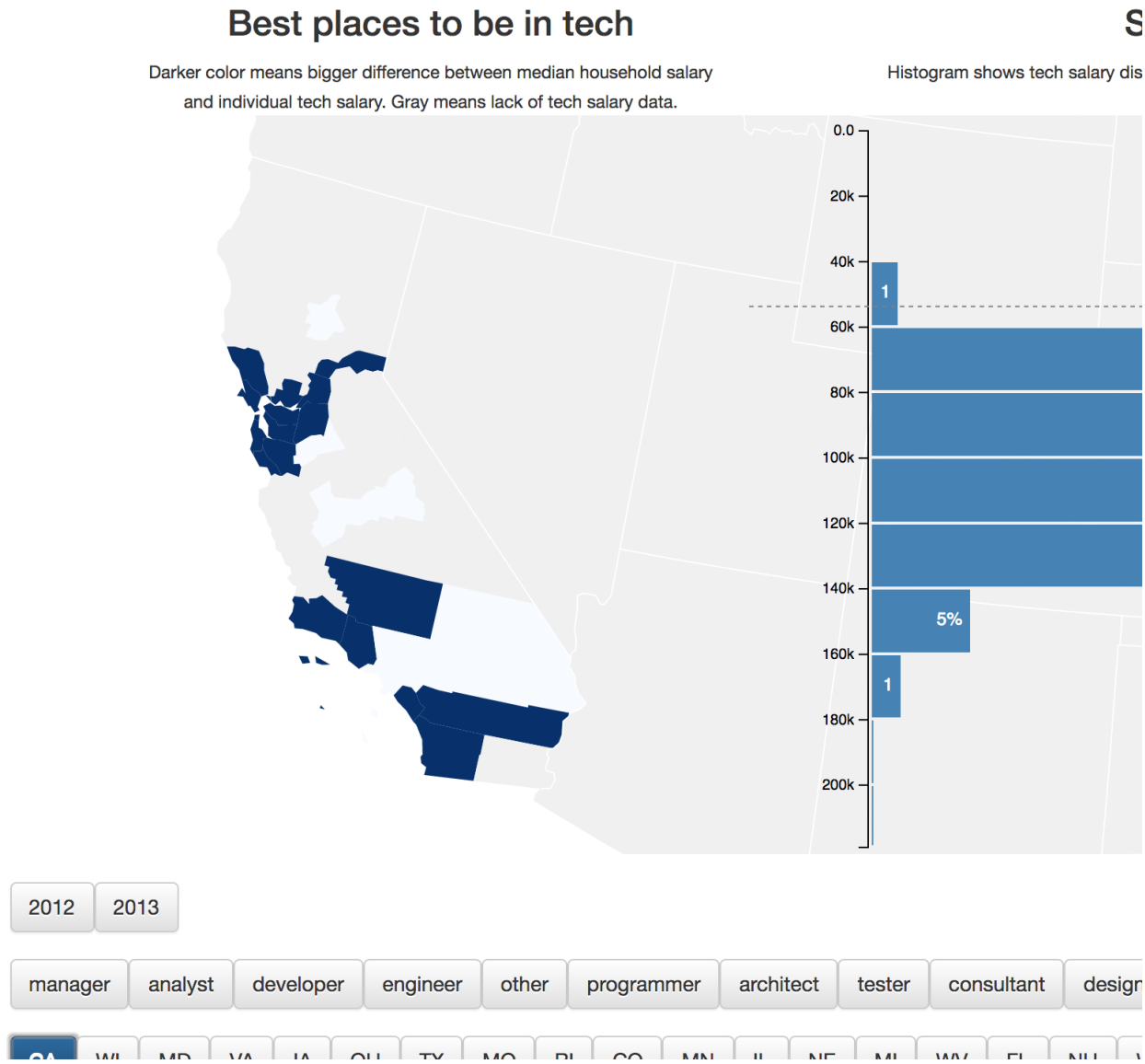
We add a `.filter` call to `filteredSalaries`, which uses our `salariesFilter` method to throw out anything that doesn't fit. Then we set up `zoom` if a US state was selected.

We built the `CountyMap` component to focus on a given US state. Finding the centroid of a polygon, re-centering the map, and increasing the sizing factor. It creates a nice zoom effect.



Zoom effect

And here's the downside of this approach. SVG doesn't know about element boundaries. It just renders stuff.



Zoom without white rectangle

See, it goes under the histogram. Let's fix that and add the `Controls` render while we're at it.

Add opaque background to histogram

```
// src/App.js
class App extends Component {
    // ...

    render() {
        // ...

        return (
            <div //...>
                <svg //...>
                    <CountyMap //... />

                    <rect x="500" y="0"
                          width="600"
                          height="500"
                          style={{fill: 'white'}} />

                    <Histogram //... />
                    <MedianLine //.. />
                </svg>

                <Controls data={this.state.techSalaries}
                          updateDataFilter={this.updateDataFilter.bind(this)} />
            </div>
        )
    }
}
```

Rectangle, `500` to the right, `0` from top, `600` wide and `500` tall, with a white background. It gives the histogram an opaque background, so it doesn't matter what the map is doing.

We render the `Controls` component just after `</svg>` because it's not an SVG component – it uses normal HTML. Unlike the other components, it needs our entire dataset as `data`. We use the `updateDataFilter` prop to say which callback function it should call when a new filter is ready.

If this seems roundabout, I guess it kinda is. But it makes our app easier to componentize and keeps the code relatively unmessy. Imagine putting everything we've done so far in `App`. What a mess that'd be! :laughing:

Step 2: Build Controls component

The `Controls` component builds our filter function and `filteredBy` dictionary based on what the user clicks.

It renders 3 rows of controls and builds filtering out of the singular choice each row reports. That makes the `Controls` component kind of repetitive and a leaky abstraction to boot.

In theory, it would be better for each `ControlRow` to return a function and for `Controls` to build a composed function out of them. It's a better abstraction, but I it's think harder to understand.

We can live with a leaky abstraction and repetitive code, right? :smile:

To keep the book from getting repetitive, we're going to build everything for a `year` filter first. Then I'll show you how to add `USstate` and `jobTitle` filters as well. Once you have one working, the rest is easy.

Make a `Controls` directory in `src/components/` and let's begin. The main `Controls` component goes in the `index.js` file.

Stub Controls

Controls stubbed for year filter

```
// src/components/Controls/index.js
import React, { Component } from 'react';
import _ from 'lodash';

import ControlRow from './ControlRow';

class Controls extends Component {
    state = {
        yearFilter: () => true,
        year: '*'
    }

    updateYearFilter(year, rest) {
    }

    componentDidUpdate() {
        this.reportUpdateUpTheChain();
    }

    reportUpdateUpTheChain() {
    }
```

```
22
23      shouldComponentUpdate(nextProps, nextState) {
24          return !_.isEqual(this.state, nextState);
25      }
26
27      render() {
28      }
29  }
30
31  export default Controls;
```

We start with some imports and a `Controls` class. Inside, we define default `state` with an always-true `yearFilter` and an asterisk for `year`.

We also need an `updateYear` function, which we'll use to update the filter, a `reportUpdateUpTheChain` function called in `componentDidUpdate`, a `shouldComponentUpdate` check, and a `render` method.

Yes, we could have put everything in `reportUpdateUpTheChain` into `componentDidUpdate`. It's separate because the name is more descriptive that way. I was experimenting with some optimizations that didn't pan out, but I decided to keep the name.

I'll explain how it works and why we need `shouldComponentUpdate` after we implement the logic.

Filter logic

Year filtering logic in Controls

```
1   // src/components/Controls/index.js
2   class Controls extends Component {
3       // ...
4
5       updateYearFilter(year, reset) {
6           let filter = (d) => d.submit_date.getFullYear() === year;
7
8           if (reset || !year) {
9               filter = () => true;
10              year = '*';
11          }
12
13          this.setState({yearFilter: filter,
14                         year: year});
15      }
```

```
16
17      // ...
18
19      reportUpdateUpTheChain() {
20          this.props.updateDataFilter(
21              ((filters) => {
22                  return (d) =>  filters.yearFilter(d);
23              })(this.state),
24              {
25                  year: this.state.year
26              }
27          );
28      }
29
30      // ...
31  }
```

When a user picks a year, the `ControlRow` components calls our `updateYearFilter` function where we build a new partial filter function. The `App` component uses it inside a `.filter` call, so we have to return `true` for elements we want to keep and `false` for elements we don't.

Comparing `submit_date.getFullYear()` with `year` achieves that.

We use the `reset` argument to reset filters back to defaults, which allows users to unselect an option.

When we have the `year` and `filter`, we update component state with `this.setState`. This triggers a re-render and calls the `componentDidUpdate` method, which calls `reportUpdateUpTheChain`.

`reportUpdateUpTheChain` then calls `this.props.updateDataFilter`, which is a callback method on `App`. We defined it earlier – it needs a new filter method and a `filteredBy` dictionary.

The code looks tricky because we're playing with higher order functions. We're making a new arrow function that takes a dictionary of filters as an argument and returns a new function that `&&`s them all. We invoke it immediately with `this.state` as the argument.

It looks silly when there's just one filter, but I promise it makes sense.

Now, because we used `this.setState` to trigger a callback up component stack, and because that callback triggers a re-render in `App`, which might trigger a re-render down here... because of that, we need `shouldComponentUpdate`. It prevents infinite loops. React isn't smart enough on its own because we're using complex objects in `state`.

> JavaScript's equality check compares objects on the reference level. So `{a: 1} == {a: 1}` returns `false` because the operands are different objects even though they look the same.

Render

Great, we have the logic. We should render the rows of controls we've been talking about.

Render the year ControlRow

```
// src/components/Controls/index.js
class Controls extends Component {
    // ...

    render() {
        const data = this.props.data;

        const years = new Set(data.map(d => d.submit_date.getFullYear()));

        return (
            <div>
                <ControlRow data={data}
                            toggleNames={Array.from(years.values())}
                            picked={this.state.year}
                            updateDataFilter={this.updateYearFilter.bind(this)}
                            />
            </div>
        )
    }
}
```

This is once more generalized code, but it's used for a single example: the `year` filter.

We build a `Set` of years in our dataset, then render a `ControlRow` using props to give it our `data`, a set of `toggleNames`, a callback to update the filter, and which entry is `picked` right now. This enables us to maintain the data-flows-down, events-bubble-up architecture we've been using.

If you don't know about `Sets`, they're new ES6 data structures that ensure every entry is unique. Just like a mathematical set. They're supposed to be pretty fast.

Step 3: Build ControlRow component

Now let's build the `ControlRow` component. It renders a row of controls and ensures that only one at a time is selected.

We'll start with a stub and go from there.

ControlRow stub

```
// src/components/Controls/ControlRow.js
import React, { Component } from 'react';
import _ from 'lodash';

import Toggle from './Toggle';

class ControlRow extends Component {
    componentWillMount() {
    }

    componentWillReceiveProps(nextProps) {
    }

    makePick(picked, newState) {
    }

    _addToggle(name) {
    }

    render() {
    }
}

export default ControlRow;
```

We start with imports, big surprise, then make a stub with 5 methods. Can you guess what they are?

- `componentWillMount` sets up some initial state that needs props
- `componentWillReceiveProps` calls `makePick` if a pick is set from above
- `makePick` is the `Toggle` click callback
- `_addToggle` is a rendering helper method
- `render` renders a row of buttons

State setup

```
// src/components/Controls/ControlRow.js
class ControlRow extends Component {
    // ...

    componentWillMount() {
        let toggles = this.props.toggleNames,
            toggleValues = _.zipObject(
                toggles,
                toggles.map((name) => name === this.props.picked)
            );

        this.setState({toggleValues: toggleValues});
    }

    componentWillReceiveProps(nextProps) {
        if (this.props.picked !== nextProps.picked) {
            this.makePick(nextProps.picked, true);
        }
    }

    // ...
}
```

React triggers the `componentWillMount` lifecycle hook right before it first renders our component. Mounts it into the DOM, if you will. This is a opportunity for any last minute state setup.

We take the list of `toggleNames` from props and use Lodash's `zipObject` function to create a dictionary that we save in `state`. Keys are toggle names, and values are booleans that tell us whether a particular toggle is currently picked.

You might think this is unnecessary, but it makes our app faster. Instead of running the comparison function for each toggle on every render, we build the dictionary, then perform quick lookups when rendering. Yes, `===` is a fast operator even with the overhead of a function call, but what if it was more complex?

Using appropriate data structures is a good habit. :smile:

In `componentWillReceiveProps`, we check if the `picked` value has changed, and if it has, we call `makePick` to mimic user action. This allows global app state to override local component state. It's what you'd expect in a unidirectional data flow architecture like the one we're using.

makePick implementation

```
// src/components/Controls/ControlRow.js
class ControlRow extends Component {
    makePick(picked, newState) {
        let toggleValues = this.state.toggleValues;

        toggleValues = _.mapValues(
            toggleValues,
            (value, key) => newState && key == picked // eslint-disable-line
        );

        // if newState is false, we want to reset
        this.props.updateDataFilter(picked, !newState);

        this.setState({toggleValues: toggleValues});
    }

    // ..
}
```

`makePick` changes `state.toggleValues` when the user clicks a toggle. It takes two arguments: a toggle name and the new value.

We use Lodash's `mapValues` to iterate the `name: boolean` dictionary and construct a new one with updated values. Everything that isn't `picked` gets set to `false`, and the one `picked` item becomes `true` if `newState` is `true`.

You're right if you think this is unnecessary. We could have just changed the current picked element to `false` and the new one to `true`. But I'm not entirely certain React would pick up on that. Play around and test it out :)

Next, we have a case of a misleading comment. We're calling `props.updateDataFilter` to communicate filter changes up the chain. The comment is talking about `!newState` and why it's not `newState`. → because the 2nd argument in `updateDataFilter` is called `reset`. We're only resetting filters if `newState` is false since that means a toggle was unclicked without a new one being selected.

Does that make sense? It's hard to explain without waving my arms around.

With `this.setState`, we update state and trigger a re-render, which highlights a new button as being selected.

Render a row of controls

```
// src/components/Controls/ControlRow.js
class ControlRow extends Component {
    // ...

    _addToggle(name) {
        let key = `toggle-${name}`,
            label = name;

        if (this.props.capitalize) {
            label = label.toUpperCase();
        }

        return (
            <Toggle label={label}
                    name={name}
                    key={key}
                    value={this.state.toggleValues[name]}
                    onClick={this.makePick.bind(this)} />
        );
    }

    render() {
        return (
            <div className="row">
                <div className="col-md-12">
                    {this.props.toggleNames
                         .map(name => this._addToggle(name))}
                </div>
            </div>
        );
    }
}
```

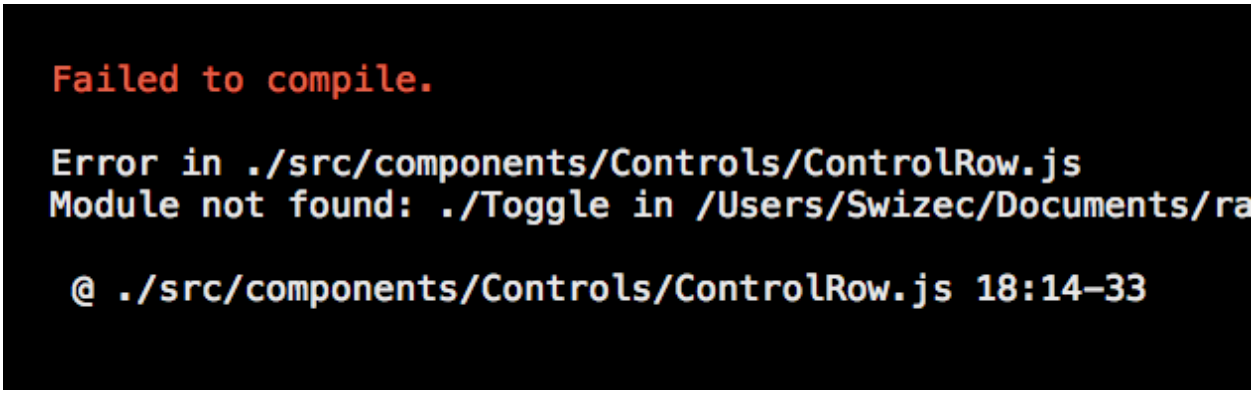
Rendering happens in two functions: `_addToggle`, which is a helper, and `render`, which is the main render.

In `render`, we set up two `divs` with Bootstrap classes. The first is a `row`, and the second is a full-width column. I tried using a column for each button, but it was annoying to manage and used too much space.

Inside the divs, we map over all toggles and use `_addToggle` to render each of them.

`_addToggle` renders a `Toggle` component with a `label`, `name`, `value` and `onClick` callback. The label is just a prettier version of the name, which also serves as a key in our `toggleValues` dictionary. It's going to be the `picked` attribute in `makePick`.

Your browser should continue showing an error, but it should change to talking about the `Toggle` component instead of `ControlRow`.



Let's build it.

Step 4: Build Toggle component

The last piece of the puzzle – the Toggle component. A button that turns on and off.

Toggle component

```
// src/components/Controls/Toggle.js
import React, { Component } from 'react';

class Toggle extends Component {
    handleClick(event) {
       this.props.onClick(this.props.name, !this.props.value);
    }

    render() {
        let className = "btn btn-default";

        if (this.props.value) {
            className += " btn-primary";
        }

        return (
            <button className={className} onClick={this.handleClick.bind(this)}>
```

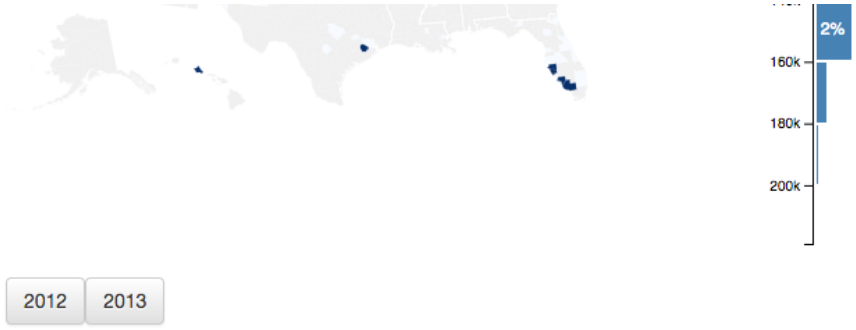
```
                {this.props.label}
            </button>
        );
    }
}

export default Toggle;
```

As always, we start with the imports, and extend React's base Component to make a new one. Small components like this are often perfect candidates for functional stateless components, but we need the click handler.

`handleClick` calls the `onClick` callback given in props, using the `name` and `!value` to identify this button and toggle its value.

`render` sets up a Bootstrap classname: `btn` and `btn-default` make an element look like a button, and the conditional `btn-primary` makes it blue. If you're not familiar with Bootstrap classes, you should check their documentation[58].

Then we render a `<button>` tag and, well, that's it. A row of year filters appears in the browser.



A row of year filters

Our shortened dataset only spans 2012 and 2013. The full dataset goes up to 2016.

If that didn't work, consult this diff on GitHub[59].

Step 5: Add US state and Job Title filters

With all that done, we can now add two more filters: US states and job titles. Our `App` component is already set up to use them, so we just have to add them to the `Controls` component.

We'll start with the `render` method, then handle the parts I said earlier would look repetitive.

[58] http://getbootstrap.com/

[59] https://github.com/Swizec/react-d3js-step-by-step/commit/a45c33e172297ca1bbcfdc76733eae75779ebd7f

Adding two more rows to Controls

```
// src/components/Controls/index.js
class Controls extends Component {
    // ...
    render() {
        const data = this.props.data;

        const years = new Set(data.map(d => d.submit_date.getFullYear())),
              jobTitles = new Set(data.map(d => d.clean_job_title)),
              USstates = new Set(data.map(d => d.USstate));

        return (
            <div>
                <ControlRow data={data}
                            toggleNames={Array.from(years.values())}
                            picked={this.state.year}
                            updateDataFilter={this.updateYearFilter.bind(this)}
                            />

                <ControlRow data={data}
                            toggleNames={Array.from(jobTitles.values())}
                            picked={this.state.jobTitle}
                            updateDataFilter={this.updateJobTitleFilter.bind(thi\
s)} />

                <ControlRow data={data}
                            toggleNames={Array.from(USstates.values())}
                            picked={this.state.USstate}
                            updateDataFilter={this.updateUSstateFilter.bind(this\
)}
                            capitalize="true" />
            </div>
        )
    }
}
```

Ok, this part is plenty repetitive, too.

We created new sets for `jobTitles` and `USstates`, then we rendered two more `ControlRow` elements with appropriate attributes. They get `toggleNames` for building the buttons, `picked` to know which is active, an `updateDataFilter` callback, and we tell US states to render capitalized.

The implementations of those `updateDataFilter` callbacks follow the same pattern as `updateYearFilter`.

New updateDataFilter callbacks

```
// src/components/Controls/index.js
class Controls extends Component {
    state = {
        yearFilter: () => true,
        year: '*',
        jobTitleFilter: () => true,
        jobTitle: '*',
        USstateFilter: () => true,
        USstate: '*'
    }

    // ...
    updateJobTitleFilter(title, reset) {
        let filter = (d) => d.clean_job_title === title;

        if (reset || !title) {
            filter = () => true;
            title = '*';
        }

        this.setState({jobTitleFilter: filter,
                       jobTitle: title});
    }

    updateUSstateFilter(USstate, reset) {
        let filter = (d) => d.USstate === USstate;

        if (reset || !USstate) {
            filter = () => true;
            USstate = '*';
        }

        this.setState({USstateFilter: filter,
                       USstate: USstate});
    }
    // ...
}
```

Yes, they're basically the same as `updateYearFilter`. THe only difference is a changed `filter` function and using different keys in `setState()`.

Why separate functions then? No need to get fancy. It would've made the code harder to read.

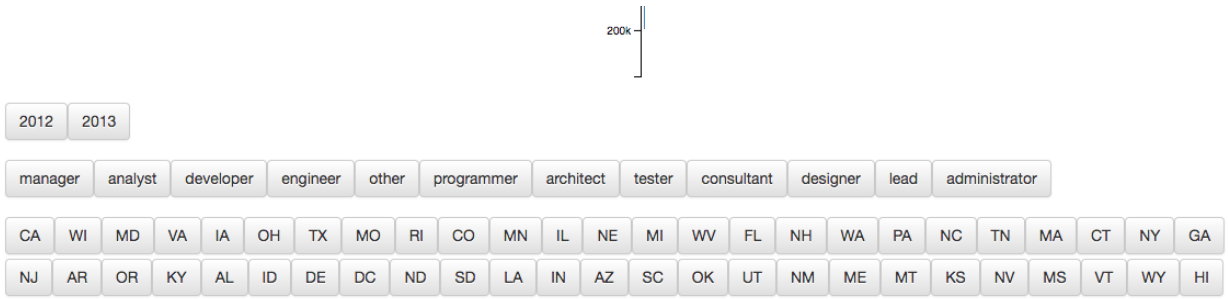
Our last step is to add these new keys to the `reportUpdateUpTheChain` function.

Add new filters to main state update

```
// src/components/Controls/index.js
class Controls extends Component {
    // ...
    reportUpdateUpTheChain() {
        this.props.updateDataFilter(
            ((filters) => {
                return (d) =>  filters.yearFilter(d)
                    && filters.jobTitleFilter(d)
                    && filters.USstateFilter(d);
            })(this.state),
            {
                year: this.state.year,
                jobTitle: this.state.jobTitle,
                USstate: this.state.USstate
            }
        );
    }
    // ...
}
```

We add them to the filter condition with `&&` and expand the `filteredBy` argument.

Two more rows of filters show up.



All the filters

:clap:

Again, if it didn't work, consult the diff on GitHub[60].

[60] https://github.com/Swizec/react-d3js-step-by-step/commit/a45c33e172297ca1bbcfdc76733eae75779ebd7f

A small speed optimization

We're expecting a big dataset, and we're recalculating our data *and* redrawing hundreds of map elements all the time. We can fix this situation with a carefully placed `shouldComponentUpdate` to avoid updates when it shouldn't.

It goes in the main `App` component and performs a quick check for changes in the filters.

shouldComponentUpdate in App.js

```
// src/App.js
class App extends Component {
    // ...
    shouldComponentUpdate(nextProps, nextState) {
        const { techSalaries, filteredBy } = this.state;

        const changedSalaries =
        (techSalaries && techSalaries.length)
            !== (nextState.techSalaries
                && nextState.techSalaries.length);

        const changedFilters = Object.keys(filteredBy)
                                     .some(
                                         k => filteredBy[k]
                                         !== nextState.filteredBy[k]
                                     );

        return changedSalaries || changedFilters;
    }
    // ...
}
```

We take current salaries and filters from `state` and compare them with future state, `nextState`. To guess changes in the salary data, we compare lengths, and to see changes in filters, we compare values for each key.

This comparison works well enough and makes the visualization faster by avoiding unnecessary re-renders.

You shouldn't really notice any particular change with the shortened dataset, but if things break, consult the diff on Github[61].

[61] https://github.com/Swizec/react-d3js-step-by-step/commit/b58218eb2f18d6ce9a789808394723ddd433ee1d

Rudimentary routing

Imagine this. A user finds your dataviz, clicks around, and finds an interesting insight. They share it with their friends.

"See! I was right! This link proves it."

"Wtf are you talking about?"

"Oh... uuuuh... you have to click this and then that and then you'll see. I'm legit winning our argument."

"Wow! Kim Kardashian just posted a new snap with her dog."

Let's help our intrepid user out and make the dataviz linkable. We should store the current `filteredBy` state in the URL and be able to restore from a link.

There are many ways to achieve this. ReactRouter[62] comes to mind, but the quickest is to implement our own rudimentary routing. We'll add some logic to manipulate and read the URL hash.

THe easiest place to put this logic is next to the existing filter logic inside the `Controls` component. Better places exist from a "low-down components shouldn't play with global stuff" perspective, but that's okay.

Adding rudimentary routing

```
// src/components/Controls/index.js
class Controls extends Component {
    // ...
    componentDidMount() {
        let [year, USstate, jobTitle] = window.location
                                            .hash
                                            .replace('#', '')
                                            .split("-");

        if (year !== '*' && year) {
            this.updateYearFilter(Number(year));
        }
        if (USstate !== '*' && USstate) {
            this.updateUSstateFilter(USstate);
        }
        if (jobTitle !== '*' && jobTitle) {
```

[62] https://github.com/ReactTraining/react-router

```
            this.updateJobTitleFilter(jobTitle);
        }
    }

    componentDidUpdate() {
        window.location.hash = [this.state.year || '*',
                                this.state.USstate || '*',
                                this.state.jobTitle || '*'].join("-");

        this.reportUpdateUpTheChain();
    }
    // ...
}
```

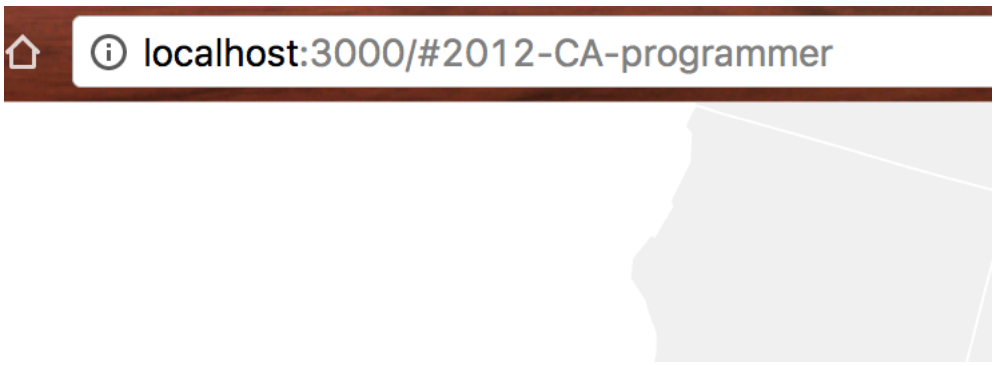
We use the `componentDidMount` lifecycle hook to read the URL when our component first renders on the page. Presumably when the page loads, but it could be later. It doesn't really matter *when*, just that we update our filter the first chance we get.

`window.location.hash` gives us the hash part of the URL and we clean it up and split it into three parts: `year`, `USstate`, and `jobTitle`. If the URL is `localhost:3000/#2013-CA-manager`, then `year` becomes `2013`, `USstate` becomes `CA`, and `jobTitle` becomes `manager`.

We make sure each value is valid and use our existing filter update callbacks to update the visualization. Just like it was the user clicking a button.

In `componentDidUpdate`, we now update the URL hash as well as call `reportUpdateUpTheChain`. Updating the hash just takes assigning a new value to `window.location.hash`.

You should now see the URL changing as you click around.



Changing URL hash

There's a bug with some combinations in 2013 that don't have enough data. It will go away when we use the full dataset.

If it doesn't work at all, consult the diff on Github[63].

[63] https://github.com/Swizec/react-d3js-step-by-step/commit/2e8fb070cbee5f1e942be8ea42fa87c6c0379a9b

Prep for launch

You've built a great visualization. Congratz! It's time to put it online and share with the world.

To do that, we're going to use Github Pages because our app is a glorified static website. There's no backend, so all we need is something to serve our HTML, JavaScript, and CSV. Github Pages is perfect for that.

It's free, works well with `create-react-app`, and can withstand a lot of traffic. You don't want to worry about traffic when your app gets to the top of HackerNews or Reddit.

There are a few things we should take care of:

- setting up deployment
- adding a page title
- adding some copy
- Twitter and Facebook cards
- an SEO tweak for search engines
- use the full dataset

Setting up deployment

You'll need a Github repository. If you're like me, writing all this code without version control or off-site backup made you nervous, so you already have one.

For everyone else, head over to Github, click the green `New Repository` button and give it a name. Then copy the commands it gives you and run them in your console.

It should be something like this:

Put code on github

```
$ git init
$ git commit -m "My entire dataviz"
$ git remote add origin git://github ...
$ git push origin -u master
```

If you've been git-ing locally without pushing, then you need only the `git remote add` and `git push origin` commands. This puts your code on Github. Great idea for anything you don't want to lose if your computer craps out.

Every Github repository comes with an optional Github Pages setup. The easiest way for us to use it is with the `gh-pages` npm module.

Install it with this command:

Install gh-pages helper

```
$ npm install --save-dev gh-pages
```

Add two lines to package.json:

Update package.json

```
// package.json
"homepage": "https://<your username>.github.io/<your repo name>"
"scripts": {
        "eject": "react-scripts eject",
        "deploy": "npm run build && gh-pages -d build"
}
```

We've been ignoring the `package.json` file so far, but it's a pretty important file. It specifies all of our project's dependencies, meta data for npm, and scripts that we run locally. This is where `npm start` is defined, for instance.

We add a `deploy` script that runs `build` and a `gh-pages` deploy, and we specify a `homepage` URL. The URL tells our build script how to set up URLs for static files in `index.html`.

Github Pages URLs follow a `https://<your username>.github.io/<your repo name>` schema. For instance, mine is `https://swizec.github.io/react-d3js-step-by-step`. Yours will be different.

You can deploy with `npm run deploy`. Make sure all changes are committed. We'll do it together when we're done setting up.

Twitter and Facebook cards and SEO

How your visualization looks on social media matters more than you'd think. Does it have a nice picture, a great description, and a title, or does it look like a random URL? Those things matter.

And they're easy to set up. No excuse.

We're going to poke around `public/index.html` for the first time. Add titles, Twitter cards, Facebook Open Graph things, and so on.

Basic SEO

```
<!-- public/index.html -->
<head>
    <!-- //... -->
    <title>How much does an H1B in tech pay?</title>

    <link rel="canonical"
          href="https://swizec.github.io/react-d3js-step-by-step/" />
</head>
<body>
    <!-- //... -->
    <div id="root">
        <h2>The average H1B in tech pays $86,164/year</h2>

        <p class="lead">
            Since 2012 the US tech industry has sponsored 176,075
            H1B work visas. Most of them paid <b>$60,660 to $111,668</b>
            per year (1 standard deviation). <span>The best city for
            an H1B is <b>Kirkland, WA</b> with an average individual
            salary <b>$39,465 above local household median</b>.
            Median household salary is a good proxy for cost of
            living in an area.</span>
        </p>
    </div>
</body>
```

We add a `<title>` and a `canonical` URL. Titles configure what shows up in browser tabs, and the canonical URL is there to tell search engines that this is the main and most important URL for this piece of content. This is especially important for when people copy-paste your stuff and put it on other websites.

But I messed up here and used the wrong URL. This knocks down rankings for the original version of this visualization, but oh well :smile:

In the body root tag, we add some copy-pasted text from our dataviz. You'll recognize the default title and description.

As soon as React loads, these get overwritten with our preloader, but it's good to have them here for any search engines that aren't running JavaScript yet. I think both Google and Bing are capable of running our React app and getting text from there, but you never know.

To make social media embeds look great, we'll use Twitter card[64] and Facebook OpenGraph[65] meta

[64] https://dev.twitter.com/cards/types/summary-large-image
[65] https://developers.facebook.com/docs/sharing/webmasters

tags. I think most other websites just rely on these since most people use them. They go in the `<head>` of our HTML.

Add FB and Twitter cards

```
<!-- public/index.html -->
<head>
    <meta property="og:locale" content="en_US" />
    <meta property="og:type" content="article" />
    <meta property="og:title"
          content="The average H1B in tech pays $86,164/year" />
    <meta property="og:description"
          content="Since 2012 the US tech industry has sponsored
176,075 H1B work visas. With an average individual salary
up to $39,465 above median household income." />
    <meta property="og:url"
          content="https://swizec.github.io/react-d3js-step-by-step" />
    <meta property="og:site_name" content="A geek with a hat" />
    <meta property="article:publisher"
          content="https://facebook.com/swizecpage" />
    <meta property="fb:admins" content="624156314" />
    <meta property="og:image"
          content="https://swizec.github.io/react-d3js-step-by-step/thumbnail.pn\
g" />

    <meta name="twitter:card" content="summary_large_image" />
    <meta name="twitter:description"
          content="Since 2012 the US tech industry has sponsored
176,075 H1B work visas. With an average individual salary
up to $39,465 above median household income." />
    <meta name="twitter:title"
          content="The average H1B in tech pays $86,164/year" />
    <meta name="twitter:site" content="@swizec" />
    <meta name="twitter:image"
          content="https://swizec.github.io/react-d3js-step-by-step/thumbnail.pn\
g" />
    <meta name="twitter:creator" content="@swizec" />
</head>
```

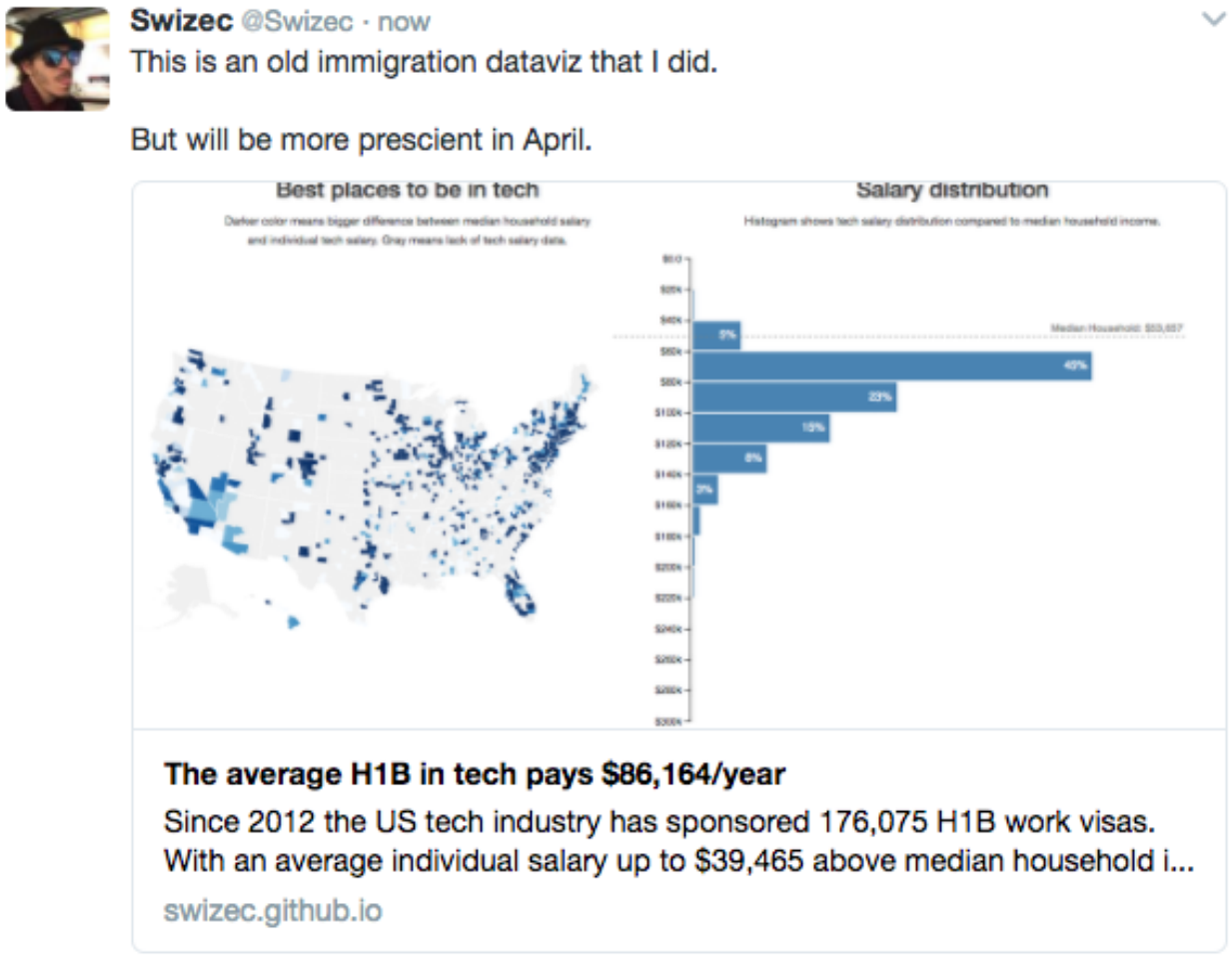
Much of this code is repetitive. Both Twitter and Facebook want the same info, but they're stubborn and won't read each other's formats. You can copy all of this, but make sure to change `og:url`, `og:site_name`, `article:publisher`, `fb:admins`, `og:image`, `twitter:site`, `twitter:image`, and `twitter:creator`. They're specific to you.

The URLs you should change to the `homepage` URL you used above. The rest you should change to use your name and Twitter/Facebook handles. I'm not sure *why* it matters, but I'm sure it does.

An important one is `fb:admin`. It enables admin features on your site if you add their social plugins. If you don't, it probably doesn't matter.

You're also going to need a thumbnail image. I made mine by taking a screenshot of the final visualization, then I put it in `public/thumbnail.png`.

Now when somebody shares your dataviz on Twitter or Facebook, it's going to look something like this:



Dataviz Twitter card

Full dataset

One more step left to do. Use the whole dataset!

Go into `src/DataHandling.js` and change one line:

Switch to full dataset

```
// src/DataHandling.js
export const loadAllData = (callback = _.noop) => {
    d3.queue()
        // ..
        .defer(d3.csv, 'data/h1bs-2012-2016-shortened.csv', cleanSalary)
        .defer(d3.csv, 'data/h1bs-2012-2016.csv', cleanSalary)
        // ...
}
```

We change the file name, and that's that. Full dataset locked and loaded. Dataviz ready to go.

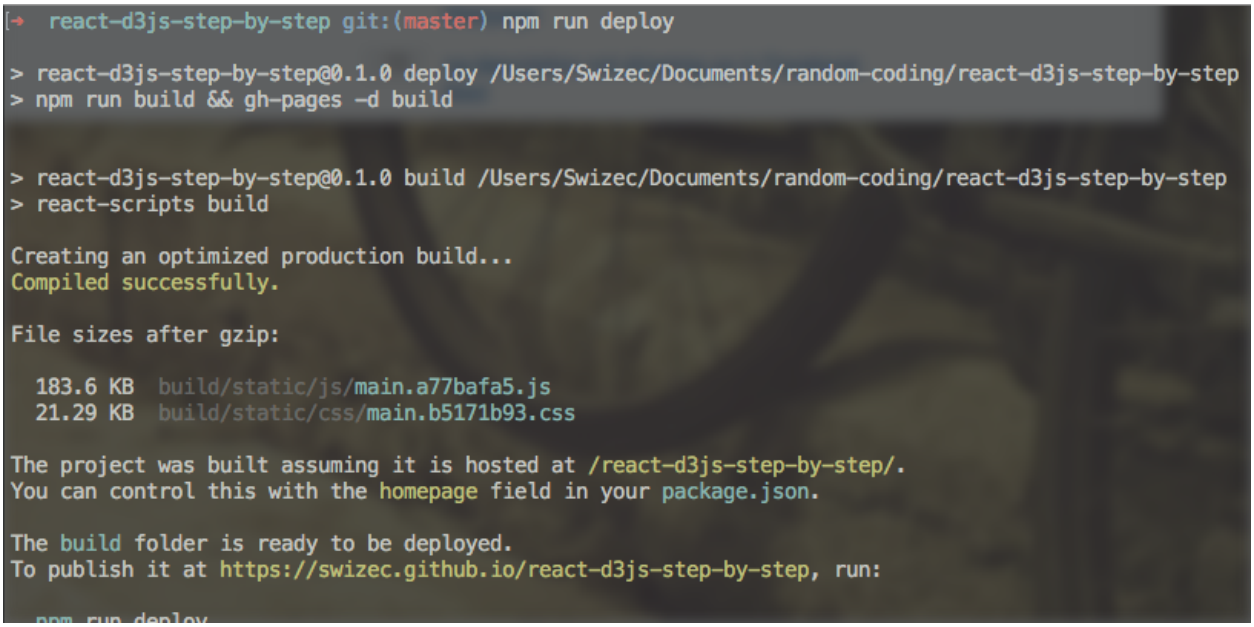
Launch!

To show your dataviz to the world, make sure you've committed all changes. Using `git status` shows you anything you've forgotten.

Then run:

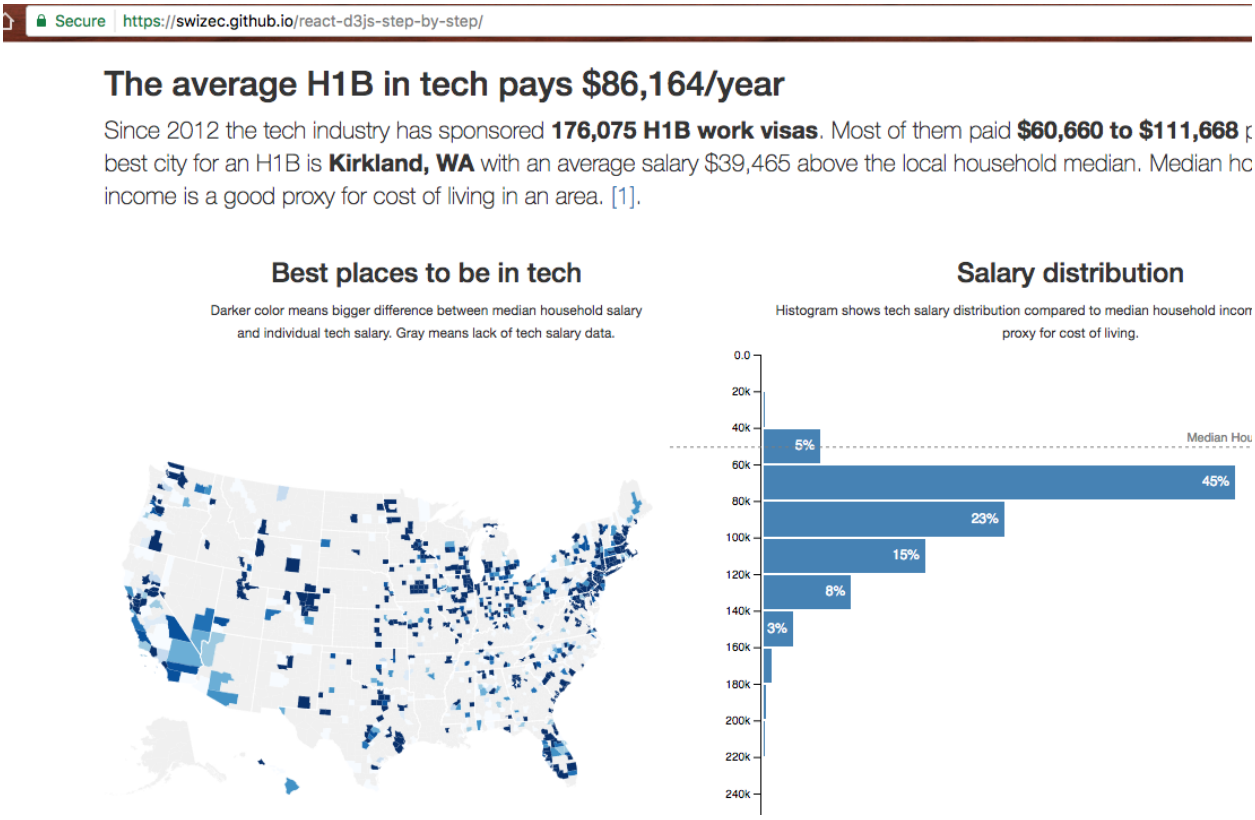
```
$ npm run deploy
```

You'll see a bunch of output:



Deploy output

And you're ready to go. Your visualization is online. My URL is `https://swizec.github.io/react-d3js-step-by-step/`, yours is different. Visit it and you'll see what you've built. Share it and others will see it too.



Deployed dataviz

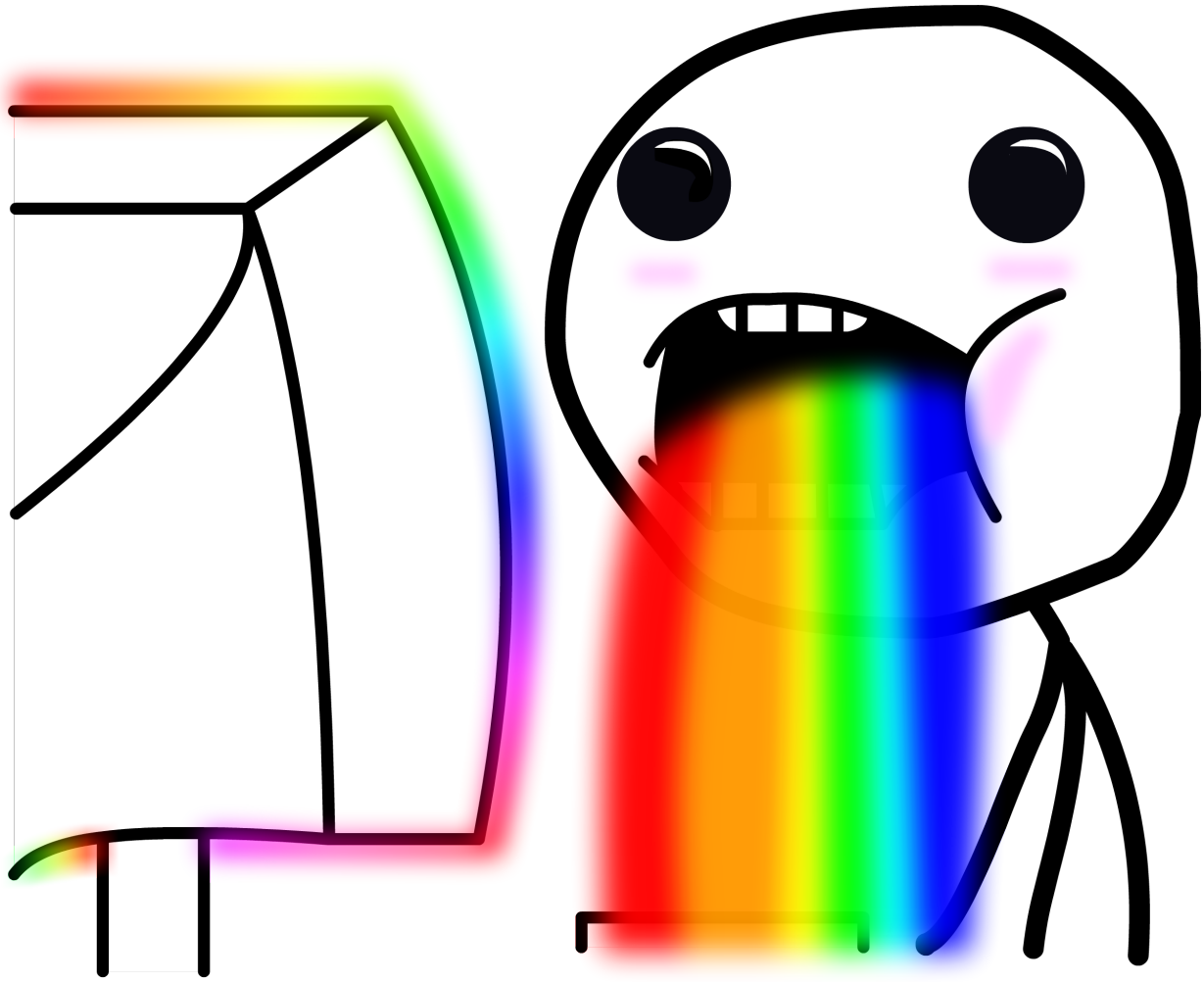
Congratz! You've built and deployed your first React and D3v4 dataviz. You're amazing \o/

Thanks for following along with the meaty part of my book. You're now well equipped to make cool things.

In the next section, we're going to look at building animations.

Animation

Welcome to the animation section. This is where the real fun begins. Demos that look cool and impress your friends.



You already know how React and D3 work together, so these demos are going to go faster. You know that we're using React for rendering SVG and D3 for calculating props. You know how to make your dataviz interactive and how to handle oodles of data.

Now you're going to learn how to make it dance. Time to build smooth transitions between states, build complex animations, and interact with the user in real-time. 60 frames per second, baby!

Our general approach to animation goes like this: Render from state → Change state 60 times per second. Animation!

We're going to use two different ways of changing state so often. The first follows a game loop principle, which gives you more control, but is more tedious. The second uses D3 transitions, which is quicker to build, but gives you less control.

We're going to start with an example or two in CodePen, then build something more involved. But don't worry, no more big huge projects like the tech salary visualization

Using a game loop for rich animation

I love game loops. It even sounds fun: “game loop”. Maybe it’s just that whenever I build a game loop, the thing I’m building is fun to play with. :thinking_face:

A game loop is an infinite loop where each iteration renders the next frame of your game or animation. You do your best to complete each iteration in 16 milliseconds, and your user gets smooth animation.

As you can imagine, our challenge is to cram all the physics and rendering into those 16 milliseconds. The more elements you’re rendering, the harder it gets.

A bouncing ball

Let’s get our feet wet with my favorite childhood example: a bouncing ball.

I must have built dozens of them back in my Turbo Pascal[66] days using BGI[67]. Yes, those Turbo Pascal and BGI are from the 80’s. No, I’m not that old. I just started young and with old equipment. Coding for DOS is easier when you’re a kid than coding for Windows 95.

Here is a screenshot of our bouncing ball:

Bouncing Ball

[66] https://en.wikipedia.org/wiki/Turbo_Pascal

[67] https://en.wikipedia.org/wiki/Borland_Graphics_Interface

Exciting, isn't it? Took me five tries to catch it. Future examples will look better as screenshots, I promise.

I suggest you follow along on CodePen. Here's one I prepared for you earlier: click me[68]

Step 1: stub it out

We start with a skeleton: An empty `Ball` component, and an `App` component stubbed out to run the game loop.

Bouncy ball skeleton

```
const Ball = ({ x, y }) => (

);

const MAX_H = 750;

class App extends Component {
  constructor() {
    super();

    this.state = {
      y: 5,
      vy: 0
    }
  }

  componentDidMount() {
    // start game loop
  }

  componentWillUnmount() {
    // stop loop
  }

  gameLoop() {
    // move ball
  }

  render() {
    // render svg
```

[68] http://codepen.io/swizec/pen/WRzqvK?editors=0010

```
  }
}
```

Nothing renders yet. CodePen complains about missing code and unexpected tokens.

The default state sets our ball's `y` coordinate to `5` and its vertical speed – `vy` – to `0`. Initial speed zero, initial position top. Perfect for a big drop.

Step 2: The Ball

We can approximate the `Ball` component with a circle. No need to get fancy; we're focusing on the animations part.

A Ball is a circle

```
const Ball = ({ x, y }) => (
  <circle cx={x} cy={y} r={5} />
);
```

Our `Ball` renders at `(x, y)` and has a radius of `5` pixels. The CSS paints it black.

It's these coordinates that we're going to play with to make the ball drop and bounce. Each time, React is going to re-render and move our ball to its new coordinates. Because we change them so quickly, it looks like the ball is animated. You'll see.

Step 3: Rendering

We need an SVG of appropriate height and a ball inside. All that goes in `App.render`.

Render ball

```
class App extends Component {
        // ...
  render() {
    return (
      <svg width="100%" height={MAX_H}>
        <Ball x={50} y={this.state.y} />
      </svg>
    )
  }
}
```

We're using `MAX_H`, which is set to 750, because a falling ball needs a lot of room to bounce up and down. You've thrown bouncy balls in a small apartment before, haven't you? It's terrifying.

A black ball should show up on your screen. Like this:

Black ball

Step 4: The Game Loop

To make the ball bounce, we need to start an infinite loop when our component first renders, change the ball's `y` coordinate on every iteration, and stop the loop when React unmounts our component. Wouldn't want to keep hogging resources, would we?

Change ball position at 60fps

```
componentDidMount() {
    this.timer = d3.timer(() => this.gameLoop());
    this.gameLoop();
}

componentWillUnmount() {
    this.timer.stop();
}

gameLoop() {
    let { y, vy } = this.state;

    if (y > MAX_H) {
```

```
14          vy = -vy*.87;
15      }
16
17      this.setState({
18          y: y+vy,
19          vy: vy+0.3
20      })
21  }
```

We start a new `d3.timer` when our App mounts, then stop it when App unmounts. This way we can be sure there aren't any infinite loops running that we can't see.

You can read details about D3 timers in d3-timer documentation[69]. The tl;dr version is that they're a lot like JavaScript's native `setInterval`, but pegged to `requestAnimationFrame`. That makes them smoother and friendlier to browser's CPU throttling features.

It basically means our game loop executes every time the browser is ready to repaint. Every 16 milliseconds, give or take.

We simulate bounce physics in the `gameLoop` function. With each iteration, we add vertical speed to vertical position and increase the speed.

Remember high school? `v = v0 + g*t`. Speed equals speed plus acceleration multiplied by time. Our acceleration is gravity, and our time is "1 frame".

And acceleration is measured in meters per second per second. Basically, the increase in speed observed every second. Real gravity is 10m/s^2, our factor is `0.3`. I discovered it when playing around. That's what looked natural.

For the bounce, we look at the `y` coordinate and compare with `MAX_H`. When it's over, we invert the speed vector and multiply with the bounce factor. Again, discovered experimentally when the animation looked natural.

Tweak the factors to see how they affect your animation. Changing the `0.3` value should make gravity feel stronger or weaker, and changing the `0.87` value should affect how high the ball bounces.

Notice that we never look at minimum height to make the ball start falling back down. There's no need. Add `0.3` to a negative value often enough and it turns positive.

Here's a CodePen with the final bouncy ball code[70].

Step 5: Correcting for time

If you run the CodePen a few times, you'll notice two bugs. The first is that sometimes our ball gets trapped at the bottom of the bounce. We won't fix this one; it's tricky.

[69] https://github.com/d3/d3-timer

[70] http://codepen.io/swizec/pen/bgvEvp?editors=0010

The second is that when you slow down your computer, the ball starts lagging. That's not how things behave in real life.

We're dealing with dropped frames.

Modern browsers slow down JavaScript in tabs that aren't focused, on computers running off battery power, when batteries get low… there's many reasons in the pile. I don't know all of them. If we want our animation to look smooth, we have to account for these effects.

We have to calculate how much time each frame took and adjust our physics.

Adjust for frame drops

```
gameLoop() {
    let { y, vy, lastFrame } = this.state;

    if (y > MAX_H) {
        vy = -vy*.87;
    }

    let frames = 1;

    if (lastFrame) {
        frames = (d3.now()-lastFrame)/(1000/60);
    }

    this.setState({
        y: y+vy*frames,
        vy: vy+0.3*frames,
        lastFrame: d3.now()
    })
}
```

We add `lastFrame` to game state and set it with `d3.now()`. This gives us a high resolution timestamp that's pegged to `requestAnimationFrame`. D3 guarantees that every `d3.now()` called within the same frame gets the same timestamp.

`(d3.now()-lastFrame)/16` tells us how many frames were meant to have happened since last iteration. Most of the time, this value will be `1`.

We use it as a multiplier for the physics calculations. Our physics should look correct now regardless of browser throttling.

Unfortunately, these fixes exacerbate the "ball stuck at bottom" bug. It happens when the ball goes below `MAX_H` and doesn't have enough bounce to get back above `MAX_H` in a single frame.

You can fix it with a flag of some sort. Only bounce, if you haven't bounced in the last N frames. Something like that.

I suggest you play with it on CodePen: click me for time-fixed bouncy ball[71]

[71] http://codepen.io/swizec/pen/NdYNKj?editors=0010

Using transitions for simple animation

Game loops are great when you need fine-grained control. But what if you just want an element to animate a little bit when a user does something? You don't care about the details; you just want a little flourish.

That's where transitions come in.

Transitions are a way to animate SVG elements by saying *"I want this property to change to this new value and take this long to do it"*. And you can use easing functions to make it look better.

I won't go into details about *why* easing functions are important, but they make movement look more natural. You can read more about it in Disney's 12 Basic Principles of Animation[72].

The two we can achieve with easing functions are:

1. Squash and Stretch
2. Slow In Slow Out

Let me show you how it works on a small example. We're drawing a field of 50 by 50 circles that "flash" when touched. The end result looks like there's a snake following your cursor.

Rainbow snake

You can play with the code on CodePen here[73]. Follow along as I explain how it works. Tweak parameters and see what happens :smile:

App

The App component only needs a `render` method that returns an SVG. Yes, that means it could've been a functional stateless component.

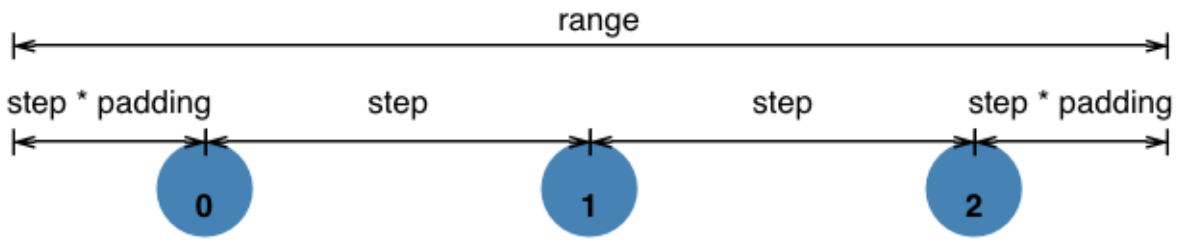
[72] https://en.wikipedia.org/wiki/12_basic_principles_of_animation
[73] http://codepen.io/swizec/pen/QdVoOg/

App render method

```
class App extends Component {
  render() {
    const width = 600,
          N = 50,
          pos = d3.scalePoint()
                  .domain(d3.range(N))
                  .range([0, width])
                  .padding(5)
                  .round(true);

    return (
      <svg width="600" height="600">
        {d3.range(N).map(x =>
           d3.range(N).map(y =>
             <Dot x={pos(x)} y={pos(y)} key={`${x}-${y}`}
                  maxPos={width} />
        ))}
      </svg>
    )
  }
}
```

We're rendering a 600px by 600px SVG, with 50 nodes per row and column. We use D3's `scalePoint` for dot positioning because it does everything we need. It makes sure they're evenly spaced, gives them padding on the sides, and ensures coordinates are rounded numbers.

Here's a diagram of how `scalePoint` works:



scalePoint diagram

To render the grid, we use two nested loops going from 0 to N. `d3.range` builds an array for us so we can `.map` over it. We return a `<Dot>` component for each iteration.

Looking at this code: `x={pos(x)} y={pos(y)}`, you can see why D3 scales are so neat. All positioning calculation boiled down to a 1-parameter function call. \o/

> Notice that unlike thus far, we didn't mess about with `updateD3` and lifecycle methods. That's useful when we're dealing with large datasets and many re-renders. You don't need the complexity when building something small.

Dot

The Dot component has more moving parts. It needs a `constructor`, a transition callback – `flash`, a `color` getter, and a `render` method.

Dot component skeleton

```
class Dot extends Component {
  constructor(props) {
    super(props);

    this.state = Object.assign({},
                               props,
                               {r: 5});
  }

  flash() {
                // transition code
  }

  get color() {
    // color calculation
  }

  render() {
    const { x, y, r, colorize } = this.state;

    return <circle cx={x} cy={y} r={r}
             ref="circle" onMouseOver={this.flash.bind(this)}
             style={{fill: colorize ? this.color : 'black'}} />
  }
}
```

We initialize state in the component `constructor`. The quickest approach is to copy all `props` to `state`, even though we don't need all props to be in state.

Normally, you want to avoid state and render all components from props. Functional principles, state is bad, and all that. But as Freddy Rangel[74] likes to say *"State is for props that change over time"*.

Guess what transitions are... props that change over time :)

So we put props in state and render from state. This lets us keep a stable platform while running transitions. It ensures that changes re-rendering `Dot` from above won't interfere with D3 transitions.

It's not *super* important in our example because those changes never happen. But I had many interesting issues in this animated typing example[75]. We'll look at that one later.

For the `render` method, we return an SVG `<circle>` element positioned at `(x, y)` with a radius, an `onMouseOver` listener, and a style with the `fill` color depending on `state.colorize`.

flash() – the transition

When you mouse over one of the dots, its `flash()` method gets called as an event callback. This is where we transition to pop the circle bigger then back to normal size.

Main Dot transition effect

```
1   flash() {
2     let node = d3.select(this.refs.circle);
3
4     this.setState({colorize: true});
5
6     node.transition()
7         .attr('r', 20)
8         .duration(250)
9         .ease(d3.easeCubicOut)
10        .transition()
11        .attr('r', 5)
12        .duration(250)
13        .ease(d3.easeCubicOut)
14        .on('end', () => this.setState({colorize: false}));
15  }
```

Here's what happens:

1. We `d3.select` the `<circle>` node. This enables D3 to take over the rendering of this particular DOM node.

[74] https://twitter.com/frangel85

[75] https://swizec.com/blog/using-d3js-transitions-in-react/swizec/6797

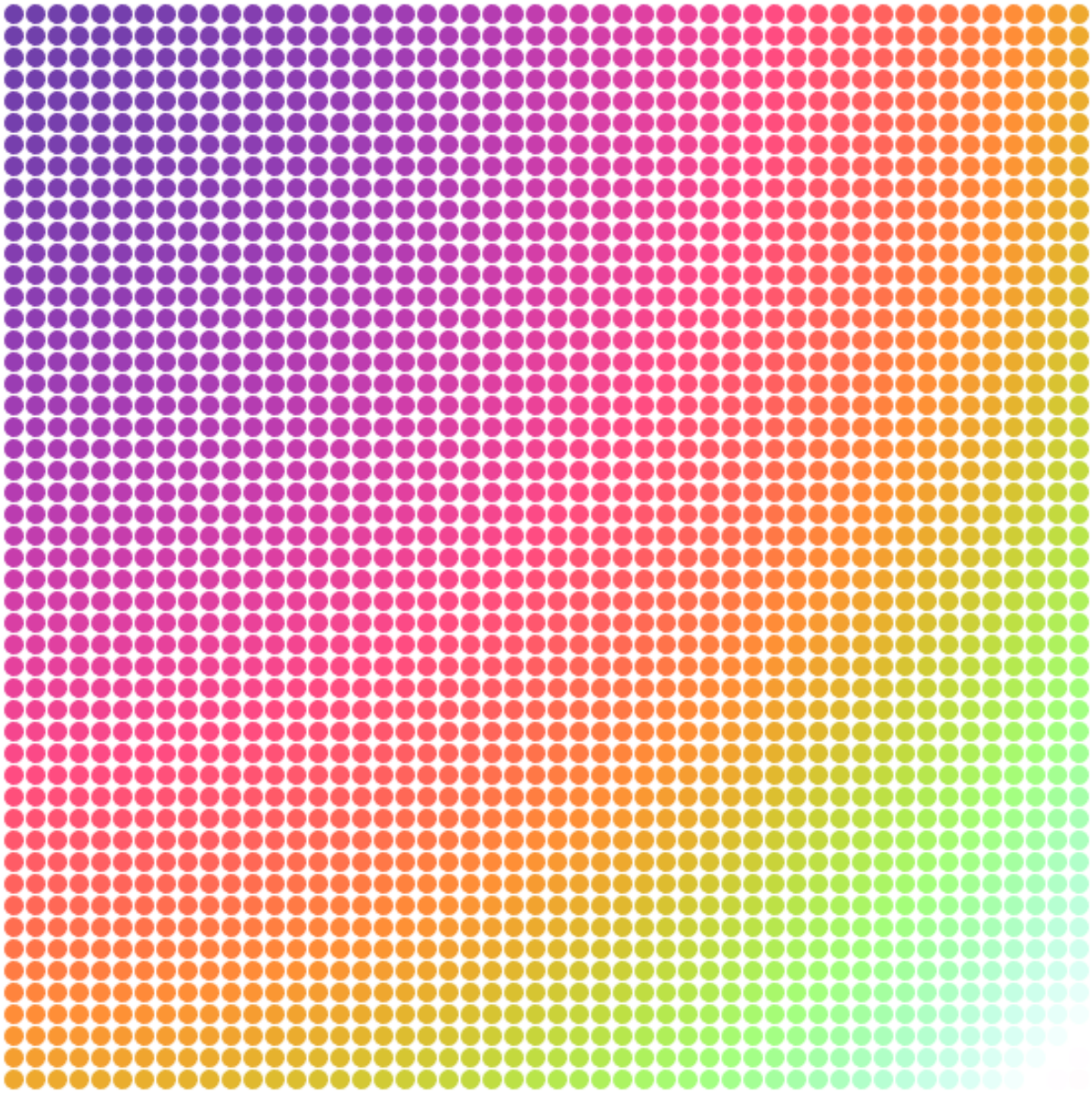
2. We `setState` to enable colorization. Yes, this triggers a re-render.
3. We start a `transition` that changes the `r` attribute to `20` pixels over a duration of `250` milliseconds.
4. We add an `easeCubicOut` easing function, which makes the animation look more natural
5. When the transition ends, we start another similar transition, but change `r` back to `5`.
6. When *that's* done, we turn off colorization and trigger another re-render.

If our transition didn't return things back to normal, I would use the `'end'` opportunity to sync React component state with reality. Something like `this.setState({r: 20})`.

get color() – the colorization

Colorization doesn't have anything to do with transitions, but I want to explain how it works. Mostly to remind you that high school math is useful.

Here's what the colored grid looks like:



Colors follow a radial pattern even though `d3.interpolateWarm` takes a single argument in the `[0, 1]` range. We achieve the pattern using circle parametrization[76].

```
x^2 + y^2 = r^2
```

Calibrate a linear scale to translate between `[0, maxR^2]` and `[0, 1]`, then feed it `x^2 + y^2`, and you get the `interpolateWarm` parameter. Magic :smile:

[76] http://www.mathopenref.com/coordparamcircle.html

Radial coloring effect

```
get color() {
  const { x, y, maxPos } = this.state;

  const t = d3.scaleLinear()
              .domain([0, 1.2*maxPos**2])
              .range([0, 1]);

  return d3.interpolateWarm(t(x**2 + y**2));
}
```

We calibrate the `t` scale to `1.2*maxPos**2` for two reasons. First, you want to avoid square roots because they're slow. Second, adding the `1.2` factor changes how the color scale behaves and makes it look better.

At least I think so. Experiment :wink:

More things to try

There's a bunch of things you can improve about this example. I suggest you try them and see how it goes.

We really should have taken the `r` parameter as a property on `<Dot>`, saved it in state as a, say, `baseR`, then made sure the transition returns our dot back to that instead of a magic `5` number. Peppering your code with magic numbers is often a bad idea.

Another improvement could be rendering more circles to provide a tighter grid. That doesn't work so well on CodePen, however. Breaks down with so many nodes. :disappointed:

Use this link to see a video[77]

You could also add transitions to the first time a circle renders. But you need something called `ReactTransitionGroup` to achieve that, and it breaks down with this many elements.

Because enter/exit transitions make many data visualizations better, we're going to look at another example just for that. A typing animation where letters fly in and fall out as you type.

[77] https://twitter.com/Swizec/status/829590239458922496

Enter/update/exit animation

Now that you know how to use transitions, it's time to take it up a notch. Enter/exit animation.

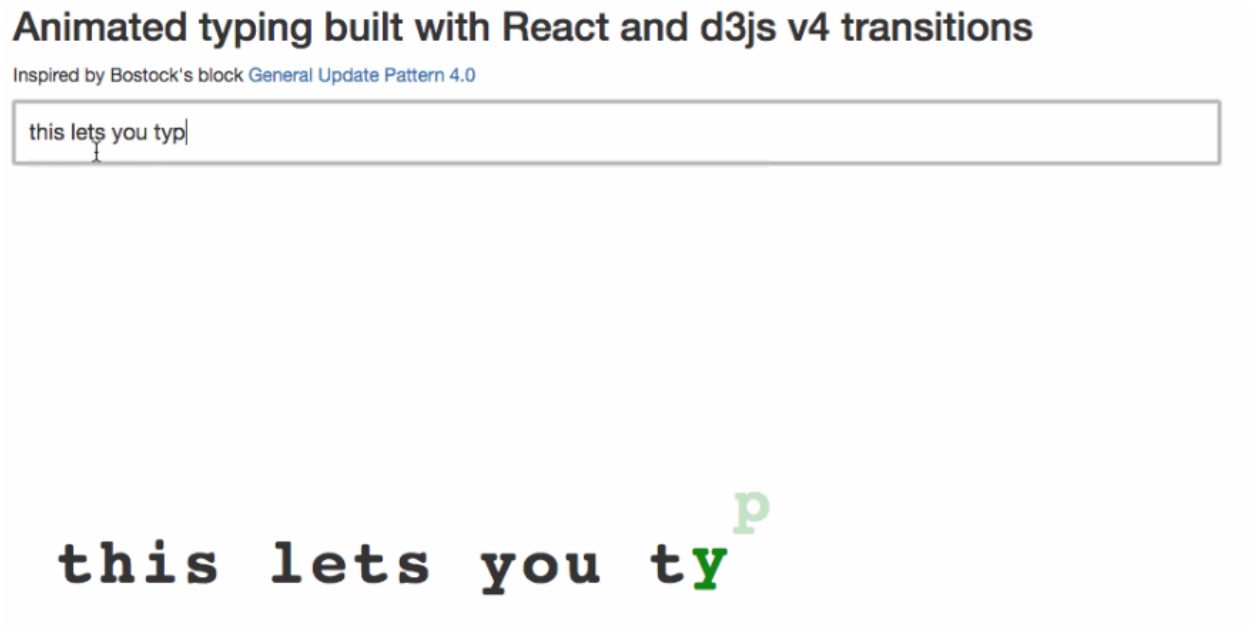
Enter/exit animations are the most common use of transitions. They're what happens when a new element enters or exits the picture. For instance, in visualizations like this famous Nuclear Detonation Timeline[78] by Isao Hashimoto. Each new Boom! flashes to look like an explosion.

I don't know how Hashimoto did it, but with React & D3v4, you'd do it with enter/exit transitions.

Another favorite of mine is this animated alphabet[79] example by Mike Bostock, the creator of D3, that showcases enter/update/exit transitions.

That's what we're going to build: An animated alphabet. New letters fall down and are green, updated letters move right or left, and deleted letters are red and fall down.

You can play with a more advanced version here[80]. Same principle as the alphabet, but it animates your typing.

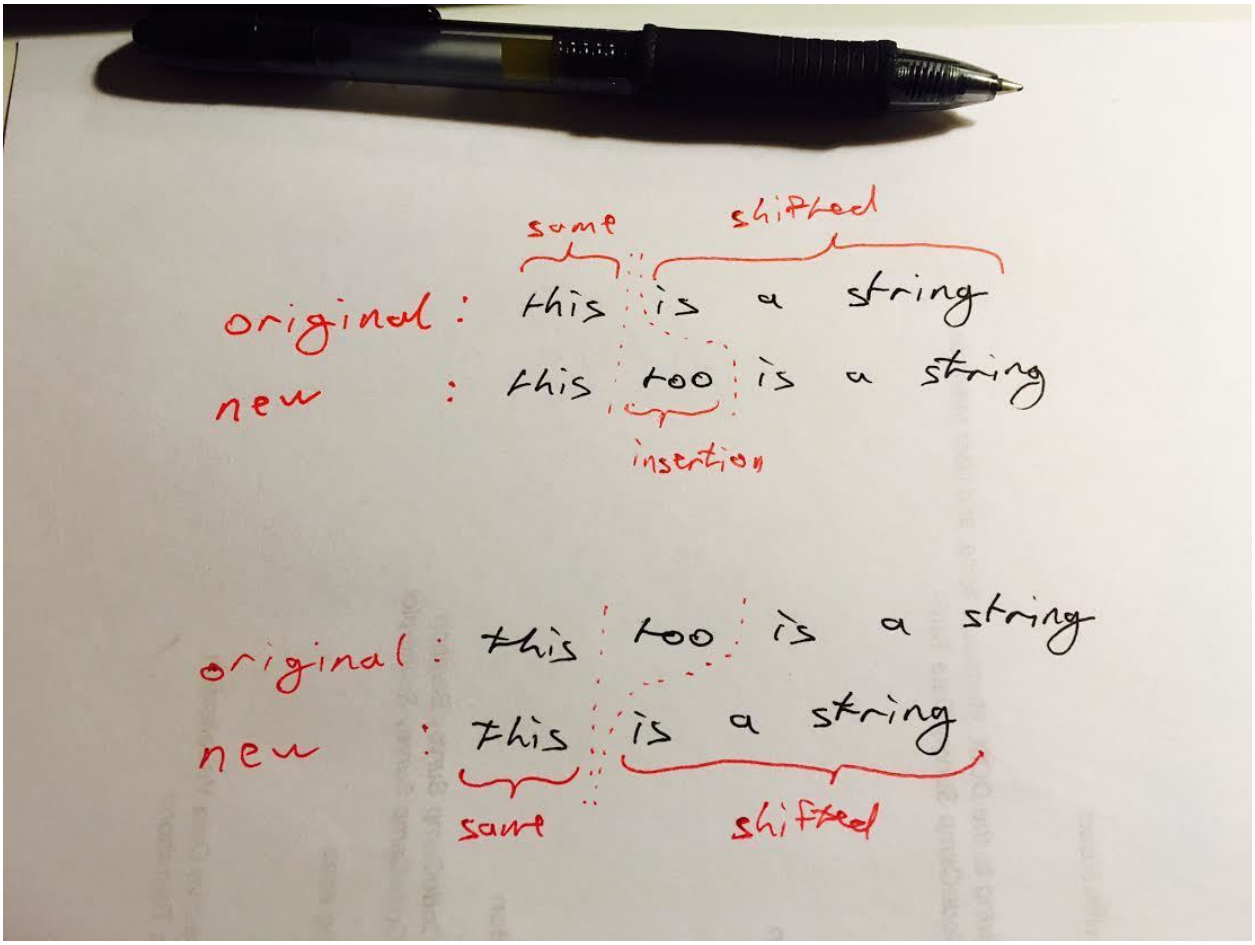


Typing animation screenshot

I wish I could embed a gif… it's 2017, and this is an electronic book, and I still can't embed animations. Silly, isn't it?

[78] https://www.youtube.com/watch?v=LLCF7vPanrY
[79] https://bl.ocks.org/mbostock/3808234
[80] http://swizec.github.io/react-d3-enter-exit-transitions/

We're building the alphabet version because the string diffing algorithm[81] is a pain to explain. I learned that the hard way when giving workshops on React and D3...



String diffing algorithm sketch

See? Easy on paper, but the code is long and weird. That, or I'm bad at implementing it. Either way, it's too tangential to explain here. You can read the article on it[82].

Animated alphabet

Our goal is to render a random subset of the alphabet. Every time the set updates, old letters transition out, new letters transition in, and updated letters transition into a new position.

We need two components:

- `Alphabet`, which creates random lists of letters every 1.5 seconds, then maps through them to

[81] https://swizec.com/blog/animated-string-diffing-with-react-and-d3/swizec/6952
[82] https://swizec.com/blog/animated-string-diffing-with-react-and-d3/swizec/6952

render `Letter` components
- `Letter`, which renders an SVG text element and takes care of its own enter/update/exit transitions

You can see the full code on GitHub here[83].

The Alphabet component

The `Alphabet` component holds a list of letters in local state and renders a collection of `Letter` components in a loop.

We start with a skeleton like this:

Alphabet skeleton

```
// src/components/Alphabet.js
import React, { Component } from 'react';
import TransitionGroup from 'react-transition-group/TransitionGroup';
import * as d3 from 'd3';

import Letter from './Letter';

class Alphabet extends Component {
    static letters = "abcdefghijklmnopqrstuvwxyz".split('');
    state = {alphabet: []};

    componentWillMount() {
        // alphabet shuffling
    }

    render() {
        // spits out SVG
    }
}

export default Alphabet;
```

We import our dependencies and define the `Alphabet` component. It holds a list of available letters in a static `letters` property and an empty `alphabet` in local state. We'll need a `componentWillMount` and a `render` method as well.

To showcase enter-update-exit transitions, we want to create a new alphabet every couple of seconds. That's easiest to do in `componentWillMount`:

[83] https://github.com/Swizec/react-d3-enter-exit-transitions/tree/alphabet

Alphabet game loop

```
// src/components/Alphabet/index.js
    componentWillMount() {
        d3.interval(() => this.setState({
           alphabet: d3.shuffle(Alphabet.letters)
                       .slice(0, Math.floor(Math.random() * Alphabet.letters.len\
gth))
                       .sort()
        }), 1500);
    }
```

We use `d3.interval( //.., 1500)` to call a function every 1.5 seconds. It's the same as `setInterval`, but friendlier to batteries and CPUs because it pegs to `requestAnimationFrame`. On each period, we shuffle the available letters, slice out a random amount, sort them, then update component state with `setState`.

This ensures our alphabet is both random and in alphabetical order.

Starting the interval in `componentWillMount` ensures it only runs when our Alphabet is on the page. You should stop these sorts of intervals in `componentWillUnmount` in real life. It's okay to skip that step in a short experiment, but it could lead to strange behavior in real world code if your component gets mounted and unmounted several times without a page refresh.

Our declarative transitions magic starts in the `render` method.

Letter rendering

```
// src/components/Alphabet/index.js
    render() {
        let transform = `translate(${this.props.x}, ${this.props.y})`;

        return (
            <g transform={transform}>
                <ReactTransitionGroup component="g">
                    {this.state.alphabet.map((l, i) => (
                        <Letter letter={l} i={i} key={`letter-${l}`} />
                     ))}
                </ReactTransitionGroup>
            </g>
        );
    }
```

An SVG transformation moves our alphabet into the specified `(x, y)` position. We map through `this.state.alphabet` inside a `<ReactTransitionGroup>` component. I'll explain why in a bit. Inside

the loop, each `Letter` gets its current text – `letter` – and index – `i`. We *have to* define the `key` attribute based on the letter – `l`.

We assume the parent component renders `<Alphabet>` inside an `<svg>` tag.

The key property

The key property is how React identifies components. Pick wrong, and you're gonna have a bad time. I spent many hours debugging and writing workarounds before I realized that basing the key on the index was a Bad Move™. *Obviously*, you want the letter to stay constant in each component and the index to change.

That's how x-axis transitions work. You're moving the letter into a specific place in the alphabet. You'll see what I mean when we look at the `Letter` component.

ReactTransitionGroup

Wrapping our list of `Letters` in `ReactTransitionGroup` gives us fine-grained access to the component lifecycle. It's a low-level API from React add-ons that expands our kingdom of lifecycle hooks.

In addition to knowing when the component mounts, updates, and unmounts, we get access to `componentWillEnter`, `componentWillLeave`, and a few others. Notice something familiar?

`componentWillEnter` is the same as d3's `.enter()`, `componentWillLeave` is the same as d3's `.exit()`, and `componentWillUpdate` is the same as d3's `.update()`.

"The same" is a strong word – they're analogous. D3's hooks operate on entire selections – groups of components – while React's lifecycle hooks operate on each component individually. In D3, an overlord dictates what happens; in React, each component knows what to do.

That makes React code easier to understand.

`ReactTransitionGroup` gives us even more hooks[84], but we don't need them for this example. I like that in both `componentWillEnter` and `componentWillLeave` we can use the callback to explicitly say *"The transition is done. React, back to you"*.

Many thanks to Michelle Tilley for writing about `ReactTransitionGroup` on StackOverflow[85].

The Letter component

Now we're ready for the component that can transition itself into and out of a visualization… without bothering anyone else *or* confusing React.

The skeleton of our `Letter` component looks like this:

[84] https://facebook.github.io/react/docs/animation.html

[85] http://stackoverflow.com/questions/29977799/how-should-i-handle-a-leave-animation-in-componentwillunmount-in-react

Letter component skeleton

```
// src/components/Alphabet/Letter.js

import React, { Component } from 'react';
import * as d3 from 'd3';

const ExitColor = 'brown',
      UpdateColor = '#333',
      EnterColor = 'green';

class Letter extends Component {
    state = {
        y: -60,
        x: 0,
        color: EnterColor,
        fillOpacity: 1e-6
    }
    transition = d3.transition()
                   .duration(750)
                   .ease(d3.easeCubicInOut);

    componentWillEnter(callback) {
        // start enter transition, then callback()
    }

    componentWillLeave(callback) {
        // start exit transition, then callback()
    }

    componentWillReceiveProps(nextProps) {
        if (this.props.i != nextProps.i) {
           // start update transition
        }
    }

    render() {
       // spit out a <text> element
    }
};

export default Letter;
```

We start with some imports and define a `Letter` component with a default state and a default transition. Yes, it feels weird to use `state` for coordinates and styling, but they're properties that change over time. State is the right place to put them.

Defining a default transition saves us some typing later on. Shared transitions can also fix animation glitches with out-of-sync transitions. For that, we'd have to define our default transition in the `Alphabet` component, then pass it into `Letter` as part of props.

All our magic values – default/final `y` coordinate, transition properties, etc. – are good candidates for props. That would make `Alphabet` more flexible but add needless complexity to this chapter. I'll leave it as an exercise for the reader ;)

componentWillEnter

We start with the enter transition in `componentWillEnter`.

Enter transition

```
// src/components/Alphabet/Letter.js
    componentWillEnter(callback) {
        let node = d3.select(this.refs.letter);

        this.setState({x: this.props.i*32});

        node.transition(this.transition)
            .attr('y', 0)
            .style('fill-opacity', 1)
            .on('end', () => {
                this.setState({y: 0,
                               fillOpacity: 1,
                               color: UpdateColor});
                callback()
            });
    }
```

We use `d3.select` to turn our `letter` ref into a d3 selection, which enables us to manipulate the DOM using d3. Then we update `this.state.x` with the current index and letter width. The width is a value that we Just Know™.

We keep our letter's `x` coordinate in `this.state` to avoid jumpiness. The `i` prop updates on each render, but we want to transition into the new position slowly. You'll see how that works in the `componentWillReceiveProps` section.

At this point, our component is invisible and 60 pixels above the baseline. That's because of the default values for `fillOpacity` and `y` that we set earlier.

To animate our component moving down and becoming visible, we use a d3 transition.

We start a new transition with `node.transition(this.transition)`, which uses settings from the default transition we defined earlier. Then we define what/how should change with `.attr` and `.style`.

The resulting transition operates directly on the DOM and doesn't tell React what's going on.

We can sync React's imagination with reality in a "transition is over" callback using `.on('end'`. We use `setState` to update component state, and trigger the main `callback`. React now knows this letter is done appearing.

componentWillLeave

The exit transition goes in `componentWillLeave` and follows the same principle, except in reverse. It looks like this:

Leave transition

```
// src/components/Alphabet/
    componentWillLeave(callback) {
        let node = d3.select(this.refs.letter);

        this.setState({color: ExitColor});

        node.transition(this.transition)
            .attr('y', 60)
            .style('fill-opacity', 1e-6)
            .on('end', () => {
                this.setState({y: 60,
                               fillOpacity: 1e-6});
                callback()
            });
    }
```

This time, we update state to change the `color` instead of `x`. That's because `x` doesn't change.

The exit transition itself is an inverse of the enter transition. An exiting letter moves further down and becomes invisible. Once the transition is over, we update state for consistency's sake, and we tell React it can remove the component.

On second though, we might not need to update state in this case. The component goes bye-bye anyway...

componentWillReceiveProps

The update transition goes into `componentWillReceiveProps` like this:

Update transition

```
// src/components/Alphabet/Letter.js
    componentWillReceiveProps(nextProps) {
        if (this.props.i !== nextProps.i) {
            let node = d3.select(this.refs.letter);

            this.setState({color: UpdateColor});

            node.transition(this.transition)
                .attr('x', nextProps.i*32)
                .on('end', () => this.setState({x: nextProps.i*32}));
        }
    }
```

You know the pattern by now, don't you? Update state, do transition, sync state with reality after transition.

In this case, we change the `color`, then we move the letter into its new horizontal position.

We could have done all of this in `componentWillUpdate` as well. However, we can't do it in `componentDidUpdate`. We need to know both the current index *and* the new index. It helps us decide whether to transition or not.

There are instances when the component updates but its horizontal position doesn't change. Every time, we call `setState` for example.

render

After all that transition magic, you might be thinking *"Holy shit, how do I render this!?"*. I don't blame ya!

But we did the hard work. Rendering is straightforward:

Letter render method

```
// src/components/Alphabet/Letter.js
    render() {
        const { x, y, fillOpacity, color } = this.state;

        return (
            <text dy=".35em"
                  x={x}
                  y={y}
                  style={{fillOpacity: fillOpacity,
```

```
                        fill: color,
                        font: 'bold 48px monospace'}}
                ref="letter">
              {this.props.letter}
            </text>
        );
    }
```

We return an SVG `<text>` element rendered at an `(x, y)` position with a `color` and `fillOpacity` style. It shows a single letter given by the `letter` prop.

As mentioned, using state for `x`, `y`, `color`, and `fillOpacity` feels weird. It's the simplest way I've found to communicate between the `render` and lifecycle methods.

That's it

Boom. We're done. You can play with a more complex version here: http://swizec.github.io/react-d3-enter-exit-transitions/[86]. Github won't let me host different branches separately.

We have an `Alphabet` component that declaratively renders an animated alphabet. Letters transition in and out and jump left and right.

All you need now is a skeleton setup that renders an SVG element and uses the `Alphabet` component.

The key takeaways are:

- use d3 for transitions
- use React to manage SVG elements
- use ReactTransitionGroup to get more lifecycle events
- mimic d3's enter/update/exit pattern

[86] http://swizec.github.io/react-d3-enter-exit-transitions/

Animating with React, Redux, and d3

And now for some pure nerdy fun: A particle generator… or, well, as close as you can get with React and D3. You'd need WebGL for a *real* particle generator.

We're making tiny circles fly out of your mouse cursor. Works on mobile with your finger, too.

To see the particle generator in action, go here[87]. Github won't let me host different branches, so you'll see the advanced 20,000 particle version from next chapter.

We're using the game loop approach to animation and Redux to store the state tree and drive changes for each frame.

You can see the full code on GitHub[88]. Code samples in this chapter focus more on the Redux part because that's what's new.

It's going to be great.

> Code in this example uses the `.jsx` file extension. I originally wrote it back when that was still a thing, and while I did update everything to React 15.5.x, I felt that changing all filenames was unnecessary.
>
> It also predates create-react-app. The code *should* work with create-react-app, but if you have issues, I suggest copying config files from GitHub[a]
>
> [a] https://github.com/Swizec/react-particles-experiment/tree/svg-based-branch

Here's how it works

We use **React to render everything**: the page, the SVG element, the particles inside. This lets us tap into React's algorithms that decide which nodes to update and when to garbage collect old nodes.

Then we use some **d3 calculations and event detection**. D3 has great random generators, so we take advantage of that. D3's mouse and touch event handlers calculate coordinates relative to our SVG. We need those, and React's click handlers are based on DOM nodes, which don't correspond to `(x, y)` coordinates. D3 looks at real cursor position on screen.

[87] http://swizec.github.io/react-particles-experiment/

[88] https://github.com/Swizec/react-particles-experiment/tree/svg-based-branch

All **particle coordinates are in a Redux store**. Each particle also has a movement vector. The store holds some useful flags and general parameters, too. This lets us treat animation as data transformations. I'll show you what I mean in a bit.

We use **actions to communicate user events** like creating particles, starting the animation, changing mouse position, and so on. On each requestAnimationFrame, we **dispatch an "advance animation" action**.

On each action, the **reducer calculates a new state** for the whole app. This includes **new particle positions** for each step of the animation.

When the store updates, **React flushes changes** via props and because **coordinates are state**, **the particles move**.

The result is smooth animation.

3 presentation components

We start with the presentation components because they're the least complicated. To render a collection of particles, we need:

- a stateless `Particle`
- a stateless `Particles`
- a proper `App`

None of them contain state, but `App` has to be a proper component so that we can use `componentDidMount`. We need it to attach D3 event listeners.

Particle

The `Particle` component is a circle. It looks like this:

Particle component

```
// src/components/Particles/Particle.jsx
import React from 'react';

const Particle = ({ x, y }) => (
        <circle cx={x} cy={y} r="1.8" />
);

export default Particle;
```

It takes `x` and `y` coordinates and returns an SVG circle.

Particles

The `Particles` component isn't much smarter – it returns a list of circles wrapped in a grouping element, like this:

Particles list

```
// src/components/Particles/index.jsx
import React from 'react';
import Particle from './Particle';

const Particles = ({ particles }) => (
        <g>{particles.map(particle =>
                <Particle key={particle.id}
                          {...particle} />
                )}
        </g>
);

export default Particles;
```

Walk through the array of particles, render a Particle component for each. Declarative rendering that you've seen before :)

We can take an array of `{id, x, y}` objects and render SVG circles. Now comes our first fun component: the `App`.

App

`App` takes care of rendering the scene and attaching d3 event listeners. It's not *really* a presentation component because it gets actions via props and ties them to mouse events. This is a Redux faux pas, but it works well in practice.

The rendering part looks like this:

App component

```
// src/components/index.jsx

import React, { Component } from 'react';
import { select as d3Select, mouse as d3Mouse, touches as d3Touches } from 'd3';


import Particles from './Particles';
import Footer from './Footer';
import Header from './Header';

class App extends Component {
        render() {
                return (
                        <div onMouseDown={e => this.props.startTicker()} style={{overflow: 'hidden
                             <Header />
                             <svg width={this.props.svgWidth}
                                  height={this.props.svgHeight}
                                  ref="svg"
                                  style={{background: 'rgba(124, 224, 249, .3)'}}>
                                 <Particles particles={this.props.particles} />
                             </svg>
                             <Footer N={this.props.particles.length} />
                         </div>
                );
        }
}

export default App;
```

There's more going on, but the gist is that we return a `<div>` with a `Header`, a `Footer`, and an `<svg>`. Inside `<svg>`, we use `Particles` to render many circles. The Header and Footer components are just some helpful text.

Notice that the core of our rendering function only says *"Put all Particles here, please"*. There's nothing about what's moved, what's new, or what's no longer needed. We don't have to worry about that.

We get a list of coordinates and naively render circles. React takes care of the rest. If you ask me, that's the real magic here.

Oh, and we call `startTicker()` when a user clicks on our scene. No reason to have the clock running *before* any particles exist.

D3 event listeners

To let users generate particles, we have to wire up some functions in `componentDidMount`. It looks like this:

Event listeners

```
// src/components/index.jsx

class App extends Component {
    componentDidMount() {
        let svg = d3Select(this.refs.svg);

        svg.on('mousedown', () => {
            this.updateMousePos();
            this.props.startParticles();
        });
        svg.on('touchstart', () => {
            this.updateTouchPos();
            this.props.startParticles();
        });
        svg.on('mousemove', () => {
            this.updateMousePos();
        });
        svg.on('touchmove', () => {
            this.updateTouchPos();
        });
        svg.on('mouseup', () => {
            this.props.stopParticles();
        });
        svg.on('touchend', () => {
            this.props.stopParticles();
        });
        svg.on('mouseleave', () => {
            this.props.stopParticles();
        });
    }

    updateMousePos() {
        let [x, y] = d3Mouse(this.refs.svg);
        this.props.updateMousePos(x, y);
    }

    updateTouchPos() {
```

```
38          let [x, y] = d3Touches(this.refs.svg)[0];
39          this.props.updateMousePos(x, y);
40      }
```

There are several events we take into account:

- `mousedown` and `touchstart` turn on particle generation
- `mousemove` and `touchmove` update the mouse location
- `mouseup`, `touchend`, and `mouseleave` turn off particle generation

Inside our event callbacks, we use `updateMousePos` and `updateTouchPos` to update Redux state. They use `d3Mouse` and `d3Touches` to get `(x, y)` coordinates for new particles relative to our SVG element and call Redux actions passed-in via props. The particle generation step uses this data as each particle's initial position.

You'll see that in the next section. Yes, it smells convoluted. We need a reference to a mouse event to get the cursor position, and we want to decouple particle generation from event handling.

Remember, React isn't smart enough to figure out mouse position relative to our drawing area. React knows that we clicked a DOM node. D3 does some magic[89] to find exact coordinates.

Touch events return lists of coordinates. One for each finger. We use only the first coordinate because shooting particles out of multiple fingers would make this example too convoluted.

That's it for rendering and user events. 107 lines of code[90].

6 Redux Actions

Redux actions are a fancy way of saying *"Yo, a thing happened!"*. They're functions you call to get structured metadata that's passed into Redux reducers.

Our particle generator uses 6 actions. The most complicated one looks like this:

Actions

```
1  // src/actions/index.js
2  export const CREATE_PARTICLES = 'CREATE_PARTICLES';
3
4  export function createParticles(N, x, y) {
5          return {
6                  type: CREATE_PARTICLES,
7                  x: x,
8                  y: y,
```

[89] https://github.com/d3/d3-selection/blob/master/src/mouse.js
[90] https://github.com/Swizec/react-particles-experiment/blob/svg-based-branch/src/components/index.jsx

```
                N: N
        };
}
```

It tells the reducer to create `N` particles at `(x, y)` coordinates. You'll see how that works when we look at the Reducer, and you'll see how it triggers when we look at the Container.

Actions *must* have a `type`. Reducers use the type to decide what to do.

Our other actions[91] are `tickTime`, `tickerStarted`, `startParticles`, `stopParticles`, and `updateMousePos`. You can guess what they mean :smile:

Personally, I think this is the least elegant part of Redux. It might make sense in big applications, but it feels unnecessary for small apps.

The whole concept smells a lot like user-defined datatypes. You give a name to a piece of shaped data. Then you use a function to generate said data. If that sounds a lot like having a named object and an object factory, or a named struct and a generator … well that's pretty much what's going on here.

¯\(☒)/¯

1 Container component

Containers are React components that talk to the Redux data store.

You can think of presentation components as templates that render stuff and containers as smart-ish views that talk to controllers. Or maybe they're the controllers.

Sometimes it's hard to tell. Basically, presentation components render and don't think, containers communicate and don't render. Redux reducers and actions think.

I'm not sure this separation is necessary in small projects. Maintaining it can be awkward and sometimes cumbersome in mid-size projects, but I'm sure it makes total sense at Facebook scale. We're using it in this project because it's the officially suggested way.

The gist of our `AppContainer` looks like this:

[91] https://github.com/Swizec/react-particles-experiment/blob/svg-based-branch/src/actions/index.js

Main container component

```
// src/containers/AppContainer.jsx
import { connect } from 'react-redux';
import React, { Component } from 'react';
import PropTypes from 'prop-types';

import App from '../components';
import { tickTime, tickerStarted, startParticles, stopParticles, updateMousePos,\
 createParticles } from '../actions';

class AppContainer extends Component {
        componentDidMount() {
                const { store } = this.context;
                this.unsubscribe = store.subscribe(() =>
                    this.forceUpdate()
                );
        }

        componentWillUnmount() {
                this.unsubscribe();
        }

   // ...

        render() {
                const { store } = this.context;
                const state = store.getState();

                return (
                    <App {...state}
                         startTicker={this.startTicker.bind(this)}
                         startParticles={this.startParticles.bind(this)}
                         stopParticles={this.stopParticles.bind(this)}
                         updateMousePos={this.updateMousePos.bind(this)}
                    />
                );
        }
};

AppContainer.contextTypes = {
    store: PropTypes.object
};
```

```
42
43 export default AppContainer;
```

We import dependencies, then we define `AppContainer` as a full-feature React `Component` because we need to use lifecycle methods. Those aren't available in functional stateless components.

Three parts of this code are important:

1. We wire up the store in `componentDidMount` and `componentWillUnmount`. Subscribe to data changes on mount, unsubscribe on unmount.
2. When rendering, we assume the store is our context, use `getState()`, then render the component we're wrapping. In this case, we render the `App` component.
3. To get the store as our context, we *have to* define `contextTypes`. It won't work otherwise.

What I like about React context is that it lets us implicitly pass properties to all children components. Anything can go in context, but if you're not careful, this can lead to rendering stale data.

That's why you should reserve context for passing the Redux store. All components need access, and the store itself is smart enough to handle updates.

AppContainer talks to the store

Congratz, you know the basics! Now we need to define those callbacks so `App` can trigger actions. Most are boilerplate-y action wrappers. Like this:

Container-store communication

```
1  // src/containers/AppContainer.jsx
2  class AppContainer extends Component {
3          // ...
4          startParticles() {
5                  const { store } = this.context;
6                  store.dispatch(startParticles());
7          }
8
9          stopParticles() {
10                 const { store } = this.context;
11                 store.dispatch(stopParticles());
12         }
13
14         updateMousePos(x, y) {
15                 const { store } = this.context;
16                 store.dispatch(updateMousePos(x, y));
```

```
        }
        // ...
}
```

Each action function gives us a `{type: ...}` object, which we dispatch on the store. You can think of dispatching as triggering a global event.

Redux uses our action, goes to all our reducers, and asks *"Anything you wanna do with this?"*. Reducers use the action type and its properties to decide how to change the state tree. You'll see how that works in the next section.

We have to look at the `startTicker` callback before we can talk about reducers. It's where our particle generator really begins.

startTicker

```
// src/containers/AppContainer.jsx
class AppContainer extends Component {
        // ...
        startTicker() {
            const { store } = this.context;

            let ticker = () => {
                if (store.getState().tickerStarted) {
                    this.maybeCreateParticles();
                    store.dispatch(tickTime());

                    window.requestAnimationFrame(ticker);
                }
            };

            if (!store.getState().tickerStarted) {
                console.log("Starting ticker");
                store.dispatch(tickerStarted());
                ticker();
            }
        }
        // ..
}
```

Don't worry if you don't "get" this immediately. I fiddled for a few hours to make it. It's our game loop!

`startTicker` creates a `ticker` function, which calls itself on each `requestAnimationFrame`. This creates an almost infinite loop.

The loop starts in two steps:

1. Check `tickerStarted` flag and start the ticker if it hasn't been started yet. This prevents running multiple game loops in parallel. As a result, we can be naive about binding `startTicker` to `onMouseDown`.
2. Create a `ticker` function that generates particles, dispatches the `tickTime` action, and calls itself on every `requestAnimationFrame`. We check the `tickerStarted` flag each time so we can stop the animation.

Yes, that means we are asynchronously dispatching redux actions. I wrote this code before Redux thunks were a thing. With those, you would package `startTicker` as an action that can dispatch other actions.

This works too, I promise :smile:

The `maybeCreateParticles` function itself isn't too interesting. It gets `(x, y)` coordinates from `store.mousePos`, checks the `generateParticles` flag – set by mousedown – and dispatches the `createParticles` action.

That's the container. 83 lines of code[92].

1 Redux Reducer

With the actions firing and the drawing done, it's time to look at the business logic of our particle generator. We'll get it done in just 33 lines of code and some change.

Well, it's a bunch of change. But the 33 lines that make up `CREATE_PARTICLES` and `TIME_TICK` changes are the most interesting. The rest is just setting various flags.

All of our logic goes in the reducer. Dan Abramov says[93] to think of reducers as the function you'd put in `.reduce()`. Given a state and a set of changes, how do I create the new state?

A "sum numbers" example would look like this:

Reducer concept

```
1  let sum = [1,2,3,4].reduce((sum, n) => sum+n, 0);
```

For each number, take the previous sum and add the number.

Our particle generator is a more complicated version of the same concept. It takes the current application state, incorporates an action, and returns the new application state.

[92] https://github.com/Swizec/react-particles-experiment/blob/svg-based-branch/src/containers/AppContainer.jsx
[93] http://redux.js.org/docs/basics/Reducers.html

To keep the example code simple, we'll put everything into the same reducer and use a big `switch` statement to decide what to do based on `action.type`. In bigger applications, we'd split our logic into domain-specific reducers. The base principle stays the same.

Let's start with the basics:

Redux reducer basic state

```
// src/reducers/index.js
const Gravity = 0.5,
          randNormal = d3.random.normal(0.3, 2),
          randNormal2 = d3.random.normal(0.5, 1.8);

const initialState = {
        particles: [],
        particleIndex: 0,
        particlesPerTick: 5,
        svgWidth: 800,
        svgHeight: 600,
        tickerStarted: false,
        generateParticles: false,
        mousePos: [null, null]
};

function particlesApp(state = initialState, action) {
        switch (action.type) {
        default:
                    return state;
        }
}

export default particlesApp;
```

This is our reducer.

We start with the gravity constant, two random generators, and define the default state:

- an empty list of particles
- a particle index, which I'll explain in a bit
- the number of particles we want to generate on each tick
- default SVG sizing
- and the two flags and `mousePos` for the generator

Our reducer doesn't change anything yet. You should always return at least the same state. Otherwise you could replace the whole thing with a big `undefined` when you don't recognize an action.

Update Redux state to animate

For most actions, our reducer updates a single value. Like this:

Reducer big switch

```
// src/reducers/index.js
function particlesApp(state = initialState, action) {
        switch (action.type) {
                case 'TICKER_STARTED':
                    return Object.assign({}, state, {
                        tickerStarted: true
                    });
                case 'START_PARTICLES':
                    return Object.assign({}, state, {
                        generateParticles: true
                    });
                case 'STOP_PARTICLES':
                    return Object.assign({}, state, {
                        generateParticles: false
                    });
                case 'UPDATE_MOUSE_POS':
                    return Object.assign({}, state, {
                        mousePos: [action.x, action.y]
                    });
                // ...
        }
}
```

Even though we're only changing values of boolean flags and two-digit arrays, *we have to create a new state*. Redux relies on application state being immutable.

Well, JavaScript doesn't have real immutability. We pretend and make sure to never change state without making a new copy first.

We use `Object.assign({}, ...` to create a new empty object, fill it with the current state, then overwrite specific values with new ones. This is fast enough even with large state trees thanks to advancements in JavaScript engines.

You can also use a library like immutable.js[94] to guarantee immutability. I haven't tried it yet.

Those state updates were boilerplate. The important ones are each tick of the game loop and creating new particles. They look like this:

[94] https://facebook.github.io/immutable-js/

The core particles logic

```
// src/reducers/index.js
function particlesApp(state = initialState, action) {
        switch (action.type) {
                // ...
    case 'CREATE_PARTICLES':
        let newParticles = state.particles.slice(0),
            i;

        for (i = 0; i < action.N; i++) {
            let particle = {id: state.particleIndex+i,
                            x: action.x,
                            y: action.y};

            particle.vector = [particle.id%2 ? -randNormal() : randNormal(),
                               -randNormal2()*3.3];

            newParticles.unshift(particle);
        }

        return Object.assign({}, state, {
            particles: newParticles,
            particleIndex: state.particleIndex+i+1
        });
    case 'TIME_TICK':
        let {svgWidth, svgHeight} = state,
            movedParticles = state.particles
                                .filter((p) =>
                                    !(p.y > svgHeight || p.x < 0 || p.x > svgW\
idth))
                                .map((p) => {
                                    let [vx, vy] = p.vector;
                                    p.x += vx;
                                    p.y += vy;
                                    p.vector[1] += Gravity;
                                    return p;
                                });

        return Object.assign({}, state, {
            particles: movedParticles
        });
                // ..
```

```
42          }
43  }
```

That looks like a bunch of code. Sort of. It's spread out.

The first part – `CREATE_PARTICLES` – copies all current articles into a new array and adds `action.N` new particles to the beginning. In my tests, this looked smoother than adding particles at the end. I don't know why. Each particle starts life at `(action.x, action.y)` and gets a random movement vector.

This randomness is another Redux faux pas. Reducers are supposed to be functionally pure: produce the same result every time they are called with the same argument values. Randomness is inherently impure.

We don't need our particle vectors to be deterministic, so I think this is fine. Let's say our universe is stochastic instead :smile:

> Stochastic means that our universe/physic simulation is governed by probabilities. You can still model such a universe and reason about its behavior. A lot of real world physics is stochastic in nature.

You could also move all this logic into the action, which would keep the reducer pure, but make it harder to see all logic in one place.

The second part – `TIME_TICK` – copies the particles array, but not each particle itself. JavaScript passes arrays by reference, which means that changing `p.vector` in a `map` mutates existing data.

This is bad from a Redux standpoint, but it's not *too* bad unless you want to use time-traveling debugging. It works faster though :smile:

We filter out any particles that have left the visible area. For the rest, we add the movement vector to their position. Then we change the `y` part of the vector using our `Gravity` constant.

That's an easy way to implement acceleration.

Our reducer is done. Our particle generator works. Our thing animates smoothly. \o/

What we learned

Building this particle generator in React and Redux, I made three important discoveries:

1. **Redux is much faster than I thought**. You'd think creating a new copy of the state tree on each animation loop was crazy, but it works. I think most our code creates only a shallow copy, which explains the speed.

2. **Adding to JavaScript arrays is slow**. Once we hit about 300 particles, adding new ones becomes visibly slow. Stop adding particles and you get smooth animation. This indicates that something about creating particles is slow: either adding to the array, or creating React component instances, or creating SVG DOM nodes.
3. **SVG is also slow**. To test the above hypothesis, I made the generator create 3000 particles on first click. The animation speed is *terrible* at first and becomes okayish at around 1000 particles. This suggests that making shallow copies of big arrays and moving existing SVG nodes around is faster than adding new DOM nodes and array elements. Here's a gif[95]

—-

There you go: Animating with React, Redux, and d3. Kind of a new superpower :wink:

Here's the recap:

- React handles rendering
- d3 calculates stuff, detects mouse positions
- Redux handles state
- element coordinates are state
- change coordinates on every `requestAnimationFrame`
- animation!

Now let's render to canvas and push this sucker to 20,000 smoothly animated elements. Even on a mobile phone.

[95] http://i.imgur.com/ug478Me.gif

Speed optimizations

Welcome to the section on speed optimizations. This is where we make our code harder to read and faster to run.

You might never need any of the techniques discussed here. You already know how to build performant data visualization components. For 99% of applications, plain code that's easy to read and understand is plenty fast.

Do you really need to save that tenth of a second at runtime and spend an extra hour of thinking every time there's a bug?

Be honest. :wink:

That said, there are cases where faster code is also easier to read. And there are cases where your visualization is so massive, that you need every ounce of oomph you can get.

For the most part, we're going to talk about three things:

- using Canvas to speed up rendering
- using React-like libraries to speed up the core library
- avoiding unnecessary computation and redraws

We'll start with Canvas because it's the biggest speed improvement you can make.

Using canvas

We've been using SVG to render our apps so far. SVG is great because it follows a familiar structure, offers infinitely scalable vector graphics, and works pretty much everywhere. Sure, you can't use some fancy SVG features on all browsers, but the basics are solid.

However, SVG has a big flaw: it's slow.

Anything more than a few hundred SVG nodes and your browser will struggle. Especially if you want to animate those thousands of elements.

A panelist at ForwardJS once asked me, *"But why would you want that?"*. It was my first time participating in a panel, and my answer sucked. What I *should* have said was, *"Because there have been thousands of UFO sightings, there are thousands of counties in the US, millions of taxi rides, hundreds of millions of people having this or that datapoint. And you want to show change over time."*

That's the real answer. Sometimes, when you're visualizing some data, you have a lot of data. The data changes over time, and animation is the best way to show changes over time.

Once upon a time, I worked on a D3 video course[96] for Packt and used UFO sightings as an example. At peak UFO sighting, right before smartphones become a thing, the animation takes up to 2 seconds to redraw a single frame.

It's terrible.

So if SVG is slow and you need to animate thousands of elements, what are you to do? HTML5 Canvas.

Why Canvas

Unlike SVG, HTML5 Canvas lets you draw rasterized images. This means you're no longer operating at the level of shapes because you're working with pixels on the screen.

With SVG, you tell the browser *what* you want to render. With Canvas, you tell the browser *how* you want to render. The browser doesn't know what you're doing; it just gets a field of pixel colors and renders them as an image.

As a result, browsers can use the GPU to render Canvas. With a bit of care, you can do almost anything you want, even on a mobile phone.

[96] https://www.packtpub.com/web-development/mastering-d3js-video

Phones these days have amazing GPUs and kind of terrible CPUs. That's why I'd almost suggest going straight to Canvas for any sort of complex animation. Mobile traffic accounts for, what, like 60% or even 70% of web traffic these days?

Now, you might be thinking this sounds complicated. How do you know which pixel should do what when you're rendering with Canvas? Sounds like a lot of work.

HTML5 Canvas does offer some shape primitives. It has circles and rectangles and things like that, but they suffer from the same problem that SVG does. The browser has to use your CPU to calculate those, and at around 10,000 elements, things break down.

Notice that 10,000 elements is still a hell of a lot more than the 3,000 or so that SVG gives you.

If your app allows it, you can use sprites: Tiny images copy-pasted on the Canvas as bytestreams. I have yet to find an upper bound for those. My JavaScript became the bottleneck :D

But I'm getting ahead of myself. We'll talk about sprites later.

The trouble with HTML5 Canvas

The tricky thing about HTML5 Canvas is that the API is low level and that canvas is flat. As far as your JavaScript and React code are concerned, it's a flat image. It could be anything.

There's no structure, which makes it difficult to detect clicks on elements, interactions between elements, when something covers something else, how the user interacts with your stuff and things like that. You have to move most of that logic into your data store and manually keep track.

As you can imagine, this becomes cumbersome. And you still can't detect user interaction because all you get is *"User clicked on coordinate (x, y). Have fun."*

At the same time, the low level API makes abstractions difficult. You can't create components for "this is a map" or "histogram goes here". You're always down to circles and rectangles and basic shapes at best.

Your code can soon start looking much like the D3.js spaghetti we wanted to avoid in the first place.

Declarative HTML5 Canvas with Konva and react-konva

Enter Konva[97] and react-konva. All the benefits of declarative code, but rendered on the canvas.

I'm gonna let Anton Lavrenov, the author of Konva, explain:

> Konva is an HTML5 Canvas JavaScript framework that enables high performance animations, transitions, node nesting, layering, filtering, caching, event handling for desktop and mobile applications, and much more.
>
> You can draw things onto the stage, add event listeners to them, move them, scale them, and rotate them independently from other shapes to support high performance animations, even if your application uses thousands of shapes. Served hot with a side of awesomeness.

That.

It's exactly what we need to push our animated React apps to thousands of elements without spending too much time thinking about the *how* of rendering. Best leave the hairy details to somebody else.

Let me show you two examples:

1. Pushing our particle generator to 20,000 elements
2. An n-body collision simulator built with MobX

A particle generator pushed to 20,000 elements with Canvas

Our SVG-based particle generator caps out at a few thousand elements. Animation becomes slow, and time between each iteration of our game loop increases.

Old elements leave the screen and get pruned faster than we can create new ones. This creates a natural (but stochastic) upper limit to how many elements we can push into the page.

[97] https://konvajs.github.io

We can render many more elements if we take out SVG and use HTML5 Canvas instead. I was able to push the code up to around 20,000 smoothly animated elements. Then JavaScript became the bottleneck.

Well, I say JavaScript was the bottleneck, but monitor size plays a role too. It goes up to 20,000 on my laptop screen, juuuust grazes 30,000 on my large desktop monitor, and averages about 17,000 on my iPhone 5SE.

Friends with newer laptops got it up to 35,000.

You can see it in action hosted on Github pages[98].

We're going to keep most of our code from before. The real changes happen in `src/components/index.jsx`, where a Konva stage replaces the `<svg>` element, and in `src/components/Particles.jsx`, where we change what we render. There's a small tweak in the `CREATER_PARTICLES` reducer to generate more particles on each tick.

If anything looks weird or doesn't work, I've prepared a helpful GitHub branch[99] for you.

You should go into your particle generator directory, install Konva and react-konva, and then make the changes below. Trying things out is better than just reading my code ;)

Install Konva

```
$ npm install --save konva react-konva
```

> react-konva is a thin wrapper on Konva itself. There's no need to think about it as its own thing. For the most part, you can go into the Konva docs, read about something, and it Just Works™ in react-konva.

Preparing a canvas layer

Our changes start in `src/components/index.jsx`. We have to throw away the `<svg>` element and replace it with a Konva stage.

You can think of a Konva stage as a Canvas element with a bunch of helper methods attached. Some of them Konva uses internally; others are exposed as an API. Functions like exporting to an image file, detecting intersections, etc.

[98] http://swizec.github.io/react-particles-experiment/
[99] https://github.com/Swizec/react-particles-experiment/tree/canvas-based-branch

Import Konva and set the stage

```
// src/components/index.jsx

// ...
import { Stage } from 'react-konva';

// ...
class App extends Component {
    // ..
    render() {
        return (
            // ..
                    <Stage width={this.props.svgWidth} height={this.props.svgHe\
ight}>
                        <Particles particles={this.props.particles} />

                    </Stage>

                </div>
                <Footer N={this.props.particles.length} />
            </div>
        );
    }
}
```

We import `Stage` from `react-konva`, then use it instead of the `<svg>` element in the `render` method. It gets a `width` and a `height`.

Inside, we render the `Particles` component. It's going to create a Konva layer and use low-level Canvas methods to render particles as sprites.

Using sprites for max redraw speed

Our SVG-based Particles component was simple. Iterate through a list of particles, render a `<Particle>` component for each.

We're going to completely rewrite it.

Our new approach goes like this:

1. Cache a sprite on `componentDidMount`
2. Clear canvas

3. Redraw all particles
4. Repeat

Because the new approach renders a flat image, and because we don't care about interaction with individual particles, we can get rid of the `Particle` component. The unnecessary layer of nesting was slowing us down.

The new `Particles` component looks like this:

Sprite-based Particles component

```
// src/components/Particles.jsx

import React, { Component } from 'react';
import { FastLayer } from 'react-konva';

class Particles extends Component {
    componentDidMount() {
        this.canvas = this.refs.layer.canvas._canvas;
        this.canvasContext = this.canvas.getContext('2d');

        this.sprite = new Image();
        this.sprite.src = 'http://i.imgur.com/m5l6lhr.png';
    }

    drawParticle(particle) {
        let { x, y } = particle;

        this.canvasContext.drawImage(this.sprite, 0, 0, 128, 128, x, y, 15, 15);
    }

    componentDidUpdate() {
        let particles = this.props.particles;

        console.time('drawing');
        this.canvasContext.clearRect(0, 0, this.canvas.width, this.canvas.height\
);

        for (let i = 0; i < particles.length; i++) {
            this.drawParticle(particles[i]);
        }
        console.timeEnd('drawing');
    }

```

```
    render() {
        return (
            <FastLayer ref="layer" transformsEnabled="position" listening="false\
">

            </FastLayer>
        );
    }
}

export default Particles;
```

40 lines of code is a lot to look at all at once. Let's walk through it, step by step.

componentDidMount

componentDidMount code

```
// src/components/Particles.jsx

// ...
componentDidMount() {
    this.canvas = this.refs.layer.canvas._canvas;
    this.canvasContext = this.canvas.getContext('2d');

    this.sprite = new Image();
    this.sprite.src = 'http://i.imgur.com/m5l6lhr.png';
}
```

This is a lifecycle hook that React calls when our component first renders. We use it to set up 3 instance properties.

`this.canvas` is a reference to the HTML5 Canvas element. We get it via a ref to the Konva layer, then spelunk through properties to get the canvas itself. As you can see by the _ prefix, Anton Lavrenov did not intend this to be a public API.

Thanks to JavaScript's permissiveness, we can use it anyway.

`this.canvasContext` is a reference to our canvas's CanvasRenderingContext2D. It's the interface we use to draw basic shapes, perform transformations, etc. Context is basically the only part of canvas you ever interact with as a developer.

Why it can't be just Canvas, I don't know.

`this.sprite` is a cached image. A small minion that we are going to copy-paste all over the place as our particle. Using a combination of `new Image()` and setting the `src` property loads it into memory as an HTML5 Image object.

It looks like this:

Our minion particle

You might be thinking that it's unsafe to copy references to rendered elements into component properties like that, but it's okay. Our render function always renders the same thing, so the reference never changes. Copying them like this means less typing.

drawParticle

drawParticle code

```
// src/components/Particles.jsx

// ...
drawParticle(particle) {
    let { x, y } = particle;

    this.canvasContext.drawImage(this.sprite, 0, 0, 128, 128, x, y, 15, 15);
}
```

This function draws a single particle on the canvas. It gets coordinates from the `particle` and uses `drawImage` to copy our sprite into position.

We use the whole sprite, corner `(0, 0)` to corner `(128, 128)`. That's how big our sprite is, by the way. And we copy it to position `(x, y)` with a width and height of `15` pixels.

`drawImage` is the fastest method I've found to put pixels on canvas. I don't know why it's so fast, but here's a helpful benchmark[100] so you can see for yourself.

componentDidUpdate

[100] https://jsperf.com/canvas-drawimage-vs-putimagedata/3

componentDidUpdate code

```
// src/components/Particles.jsx

// ...
componentDidUpdate() {
    let particles = this.props.particles;

    console.time('drawing');
    this.canvasContext.clearRect(0, 0, this.canvas.width, this.canvas.height);

    for (let i = 0; i < particles.length; i++) {
        this.drawParticle(particles[i]);
    }
    console.timeEnd('drawing');
}
```

`componentDidUpdate` is where the real particle drawing happens. React calls this method every time our list of particles changes. *After* the `render` method.

Yes, we're moving rendering out of the `render` method and into `componentDidUpdate`.

Here's how it works:

1. `this.canvasContext.clearRect` clears the entire canvas from coordinate `(0, 0)` to coordinate `(width, height)`. We delete everything and make the canvas transparent.
2. We iterate our `particles` list with a `for` loop and call `drawParticle` on each element.

Clearing and redrawing the canvas is faster than moving individual particles. For loops are faster than `.map` or any other form of iteration. I tested.

You can open your browser console to see how long it takes to draw a frame. That's why I left `console.time` and `console.timeEnd` in the code.

> `console.time` starts a timer and `timeEnd` stops it and prints the difference. I was super happy when someone told me about it during one of my livecoding sessions.

render

render code

```
// src/components/Particles.jsx

// ...
render() {
    return (
        <FastLayer ref="layer" transformsEnabled="position" listening="false">

        </FastLayer>
    );
}
```

After all that work, our `render` method is pretty simple. We render a Konva `FastLayer`, give it a `ref`, enable `position` transforms – I can't remember why to be honest, might be a leftover from experimenting – and turn off `listening` for mouse events.

Ideas for this combination of settings came from Konva's official performance tips[101] documentation. I don't understand Konva enough to know *why* this helps, but it makes sense when you think about it.

A `FastLayer` is faster than a `Layer`. It's in the name. Ignoring mouse events means you don't have to keep track of elements. It reduces computation.

All I know is that this was empirically the fastest solution that lead to the most particles on screen.

But why, Swizec?

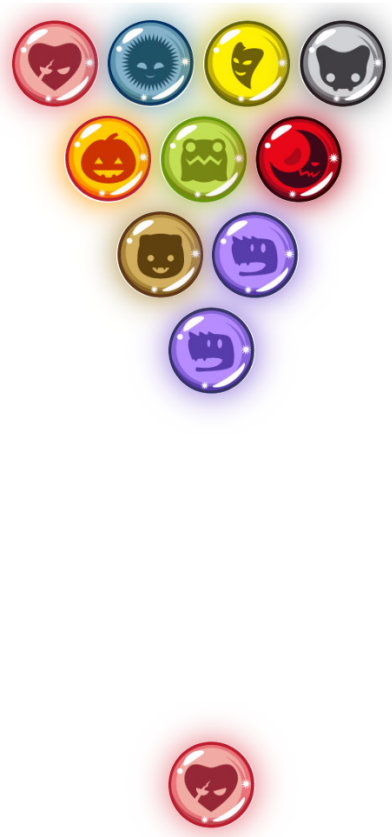
I'm glad you asked. This was a silly example. I devised the experiment because at my first React+D3 workshop somebody asked, *"What if we have thousands of datapoints, and we want to animate all of them?"*. I didn't have a good answer.

Now I do. You put them in Canvas. You drive the animation with a game loop. You are god.

You can even do it as an overlay. Have an SVG for your graphs and charts, overlay with a transparent canvas for your high speed animation.

[101] https://konvajs.github.io/docs/performance/All_Performance_Tips.html

Billiards simulation with MobX and canvas



Billiards game

We're building a small game. You have 11 glass balls – marbles, if you will. Grab one, throw it at the others, see how they bounce around. There is no point to the game, but it looks cool, and it's fun to build.

We're using React and Konva to render our 11 marbles on an HTML5 Canvas element, MobX to drive the animation loop, and D3 to help with collision detection. Because of our declarative approach to animation, we can split this example into two parts:

- Part 1: Rendering the marbles
- Part 2: Building the physics

You can see the finished code on Github[102] and play around with a hosted version[103] of the code you're about to build.

[102] https://github.com/Swizec/declarative-canvas-react-konva
[103] https://swizec.github.io/declarative-canvas-react-konva/

I know this example comes late in the book, and you're feeling like you know all there is to React and visualizations. You can think of this example as practice. It's a good way to learn the basics of MobX.

Decorators

Before we begin, let me tell you about decorators. MobX embraces them to make its API easier to use. You can use MobX without decorators, but decorators make it better. I promise.

About two years ago, decorators got very close to becoming an official spec, then were held back. I don't know *why*, but they're a great feature whose syntax is unlikely to change. So even if MobX has to change its implementation when decorators do land in the spec, you're not likely to have to change anything.

You can think of decorators as function wrappers. Instead of something like this:

Decoratorless function wrapping

```
1 inject('store', ({ store }) => <div>A thing with {store.value}</div>);
```

You can write something like this:

Function wrapping with decorators

```
1 @inject('store')
2 ({ store }) => <div>A thing with {store.value}</div>
```

I know, that's not much of a difference. It becomes better looking when you work with classes or combine multiple decorators. That's when they shine. No more `}))))}))` at the end of your function declarations.

By the way, `inject` is to MobX much like `connect` is to Redux. We'll talk more about that later.

Part 0: Some setup

Because decorators aren't in the JavaScript spec, we have to tweak how we start our project. We can still use `create-react-app`, but there's an additional step.

You should start a new project like this:

Create the billiards game project

```
1 $ create-react-app billiards-game --scripts-version custom-react-scripts
```

This command creates a new directory with a full setup for React. Just like you're used to.

The addition of `--scripts-version custom-react-scripts` employs @kitze's custom-react-scripts[104] project to give us more configuration options. Namely the ability to enable decorators.

We enable them in the `.env` file. Add this line:

Add to .env settings

```
// billiards-game/.env
// ...
REACT_APP_DECORATORS=true
```

No installation necessary. I think `custom-react-scripts` uses the `transform-decorators-legacy` Babel plugin behind the scenes. It's pre-installed, and we just enabled it with that `.env` change.

Before we begin, you should install the other dependencies as well:

Install libraries

```
$ npm install --save konva react-konva mobx mobx-react d3-timer d3-scale d3-quad\
tree
```

This installs Konva, MobX, and the parts of D3 that we need. You're now ready to build the billiards game.

A quick MobX primer

Explaining MobX in detail is beyond the scope of this book. I bet you can learn it by osmosis as you follow the code in our billiards example.

That said, here's a quick rundown of the concepts we're using.

MobX implements the ideas of reactive programming. There are values that are observable and functions that react when those values change. MobX ensures only the minimal possible set of observers is triggered on every change.

So, we have:

`@observable` – a property whose changes observers subscribe to
`@observer` – a component whose `render()` method observes values
`@computed` – a method whose value can be fully derived from obsevables
`@action` – a method that changes state, analogous to a Redux reducer
`@inject` – a decorator that injects global stores into a component's props

[104] https://github.com/kitze/custom-react-scripts

That's basically all you need to know. Once your component is an `@observer`, you never have to worry about *what* it's observing. MobX ensures it reacts only to the values that it uses during rendering.

Making your component an observer and injecting the global store is the same as using `connect` in Redux. It gives your component access to your stare, and it triggers a re-render when something that the component uses changes.

Most importantly, it *doesn't* trigger a re-render when something that the component isn't using changes. That little tidbit is what makes most other reactive libraries difficult to use.

Part 1: Rendering our marbles

Our marbles render on Canvas using Konva. Each marble is its own sprite rendered as a Konva element. This makes it easier to implement user and marble interactions.

Rendering happens in 3 components:

- `App` holds everything together
- `MarbleList` renders a list of marbles
- `Marble` renders an individual marble

We're also using 2 MobX stores:

- `Sprite` to load the marble sprite and define coordinates within
- `Physics` as our physics engine

`Sprite` and `Physics` are going to hold almost all of our game logic. We're putting a bit of drag & drop logic in the `Marble` component, but other than that, all our components are presentational. They get props and render stuff.

Let's start with `App` and work our way down.

App

Our `App` component doesn't do much. It imports our MobX stores, triggers sprite loading, and starts the game loop.

The App component

```
// src/components/App.js

import React, { Component } from 'react';
import { Provider as MobXProvider, observer } from 'mobx-react';

import Physics from '../logic/Physics';
import Sprite from '../logic/Sprite';
import MarbleList from './MarbleList';

@observer
class App extends Component {
    componentWillMount() {
        Sprite.loadSprite(() => Physics.startGameLoop());
    }

    render() {
        return (
            <div className="App">
                <div className="App-header">
                    <h2>Elastic collisions</h2>
                    <p>Rendered on canvas, built with React and Konva</p>
                </div>
                <div className="App-intro">
                    <MobXProvider physics={Physics} sprite={Sprite}>
                        <MarbleList />
                    </MobXProvider>
                </div>
            </div>
        );
    }
}

export default App;
```

We import our dependencies: React itself, a `MobXProvider` that is similar to the Redux provider (it puts stuff in react context), both of our MobX stores which export singleton instances, and the main `MarbleList` component.

`App` itself is a full featured component that initiates sprite loading in `componentDidMount` and calls `startGameLoop` when the sprite is ready. We know the sprite is ready because it calls a callback. You'll see how that works in a bit.

The `render` method outputs some descriptive text and the `MarbleList` component wrapped in a `MobXProvider`. The provider puts instances of our stores – `sprite` and `physics` – in React context.

This makes them available to all child components via the `inject` decorator.

MarbleList

Even though `MarbleList` is an important component that basically renders the whole game, it can be a functional stateless component. All the props it needs come from the two stores.

Like this:

MarbleList component

```
// src/components/MarbleList.js

import React from 'react';
import { inject, observer } from 'mobx-react';
import { Stage, Layer, Group } from 'react-konva';

import Marble from './Marble';

const MarbleList = inject('physics', 'sprite')(observer(({ physics, sprite }) =>\
 {
    const { width, height, marbles } = physics;
    const { marbleTypes } = sprite;

    return (
        <Stage width={width} height={height}>
            <Layer>
                <Group>
                    {marbles.map(({ x, y, id }, i) => (
                        <Marble x={x}
                                y={y}
                                type={marbleTypes[i%marbleTypes.length]}
                                draggable="true"
                                id={id}
                                key={`marble-${id}`} />
                    ))}
                </Group>
            </Layer>
        </Stage>
    );
}));
```

```
31
32 export default MarbleList;
```

We import our dependencies and create a `MarbleList` component. Instead of decorators, we're going with functional composition.

This shows you that MobX *can* work without decorators, but there's no real reason behind this choice. Over time, I've developed a preference to use composition for functional stateless components and decorators for class-based components.

`inject` takes values out of context and puts them in component props. `observer` declares that our component observes those props and reacts to them.

It's generally a good idea to use both `inject` and `observer` together. I have yet to find a case where you need just one or the other.

The rendering itself takes values out of our stores and returns a Konva `Stage` with a single `Layer`, which contains a `Group`. Inside this group is our list of marbles.

Each marble gets a position, a `type` that defines how it looks, an `id`, and a `key`. We set `draggable` to `true` so Konva knows that this element is draggable.

Yes, that means we get draggability on an HTML5 Canvas without any extra effort. I like that.

Marble

Each `Marble` component renders a single marble and handles dragging and dropping. We could, perhaps should, have put this logic in the physics store, but I felt this makes more sense.

The component looks like this:

Marble component

```
1  // src/components/Marble.js
2
3  import React, { Component } from 'react';
4  import { Circle } from 'react-konva';
5  import { inject, observer } from 'mobx-react';
6
7  @inject('physics', 'sprite') @observer
8  class Marble extends Component {
9      onDragStart() {
10         // set drag starting position
11     }
12
13     onDragMove() {
14         // update marble position
```

```
15      }
16
17      onDragEnd() {
18          // shoot the marble
19      }
20
21      render() {
22          const { sprite, type, draggable, id, physics } = this.props;
23          const MarbleDefinitions = sprite.marbleDefinitions;
24          const { x, y, r } = physics.marbles[id];
25
26          return (
27              <Circle x={x} y={y} radius={r}
28                      fillPatternImage={sprite.sprite}
29                      fillPatternOffset={MarbleDefinitions[type]}
30                      fillPatternScale={{ x: r*2/111, y: r*2/111 }}
31                      shadowColor={MarbleDefinitions[type].c}
32                      shadowBlur="15"
33                      shadowOpacity="1"
34                      draggable={draggable}
35                      onDragStart={this.onDragStart.bind(this)}
36                      onDragEnd={this.onDragEnd.bind(this)}
37                      onDragMove={this.onDragMove.bind(this)}
38                      ref="circle"
39                      />
40          );
41      }
42  }
43
44  export default Marble;
```

We `@inject` both stores into our component and make it an `@observer`. The `render` method takes values out of our stores and renders a Konva `Circle`. The circle uses a chunk of our sprite as its background, has a colorful shadow, and has a bunch of drag callbacks.

Those callbacks make our game playable.

In `onDragStart`, we store the starting position of the marble being dragged. In `onDragMove`, we update the marble's position in the store, which makes it possible for other marbles to bounce off ours while it's moving, and in `onDragEnd`, we shoot the marble.

Shoot direction depends on how we dragged. That's why we need to store the starting positions.

Our drag callbacks double as MobX actions, which makes our code simpler. Instead of specifying an extra `@action` in the MobX store, we manipulate the values directly.

MobX makes this okay. It keeps everything in sync and our state easy to understand. MobX even batches several value changes together before triggering re-renders.

That said, I would recommend using explicit `@actions` in larger projects.

Here's what the dragging callbacks look like.

Dragging callbacks

```
// src/components/Marble.js

class Marble extends Component {
    onDragStart() {
        const { physics, id } = this.props;

        this.setState({
            origX: physics.marbles[id].x,
            origY: physics.marbles[id].y,
            startTime: new Date()
        });
    }

    onDragMove() {
        const { physics, id } = this.props;
        const { x, y } = this.refs.circle.attrs;

        physics.marbles[id].x = x;
        physics.marbles[id].y = y;
    }

    onDragEnd() {
        const { physics } = this.props,
              circle = this.refs.circle,
              { origX, origY } = this.state,
              { x, y } = circle.attrs;


        const delta_t = new Date() - this.state.startTime,
              dist = (x - origX) ** 2 + (y - origY) ** 2,
              v = Math.sqrt(dist)/(delta_t/16); // distance per frame (= 16ms)

        physics.shoot({
           x: x,
           y: y,
           vx: (x - origX)/(v/3), // /3 is a speedup factor
```

```
            vy: (y - origY)/(v/3)
            }, this.props.id);
    }

    // ...
}
```

In `onDragStart`, we store original coordinates and start time in local state. These are temporary values that nobody outside this user action cares about. Local state makes sense.

We'll use them to determine how far the user dragged our marble.

In `onDragMove` we update the MobX stare with new coordinates for this particular marble. You might think we're messing with mutable state here, and we *might* be, but these are MobX observables. They're wrapped in setters that ensure everything is kept in sync, changes are being logged, etc.

I don't know how MobX keeps track behind the scenes. I just know it's okay to "change" values.

`onDragEnd` is where the marble shooting happens. We calculate drag speed and direction, then we call the `shoot()` action on the `physics` store.

If you're not sure how our math works, I suggest looking up euclidean distance[105].

Sprite store

Now that we know how rendering works, we need to load our sprite. It's an icon set I bought somewhere online, but I can't remember where or who from.

Here's what it looks like:

[105] https://en.wikipedia.org/wiki/Euclidean_distance

Marbles sprite

To use this, we need two things:

1. A way to tell where on the image each marble lies
2. A MobX store that loads the image into memory

The first is a `MarbleDefinitions` dictionary. We used it in `Marble` component's render method.

MarbleDefinitions dictionary

```
// src/logic/Sprite.js

const MarbleDefinitions = {
    dino: { x: -222, y: -177, c: '#8664d5' },
    redHeart: { x: -222, y: -299, c: '#e47178' },
    sun: { x: -222, y: -420, c: '#5c96ac' },

    yellowHeart: { x: -400, y: -177, c: '#c8b405' },
    mouse: { x: -400, y: -299, c: '#7d7e82' },
    pumpkin: { x: -400, y: -420, c: '#fa9801' },

    frog: { x: -576, y: -177, c: '#98b42b' },
    moon: { x: -575, y: -299, c: '#b20717' },
    bear: { x: -576, y: -421, c: '#a88534' }
};

export { MarbleDefinitions };
```

Each type of marble has a name, a coordinate, and a color. The coordinate tells us where on the sprite image it is, and the color helps us define a nice looking shadow.

All values were painstakingly assembled by hand. You're welcome.

The MobX store that loads our sprite into memory and helps us use it looks like this:

Sprite store

```
// src/logic/Sprite.js

import { observable, action, computed } from 'mobx';
import MarbleSprite from '../monster-marbles-sprite-sheets.jpg';

class Sprite {
    @observable sprite = null;

    @action loadSprite(callback = () => null) {
        const sprite = new Image();
        sprite.src = MarbleSprite;

        sprite.onload = () => {
            this.sprite = sprite;

```

```
            callback();
        };
    }

    @computed get marbleTypes() {
        return Object.keys(MarbleDefinitions);
    }

    @computed get marbleDefinitions() {
        return MarbleDefinitions;
    }
}

export default new Sprite();
```

As you can see, a MobX store is a JavaScript object. It has `@observable` values, `@actions`, and `@computed` getters. There's nothing more to it than that.

No complicated reducers and action generators, just JavaScript functions and properties. There's plenty going on behind the scenes, but we don't have to think about it.

That's why I personally like MobX more than Redux. Feels easier to use :smile:

In the `Sprite` store, we have an `@observable sprite`. Changing this value will trigger a re-render in our `@observer` components that rely on it. Every marble in our case.

Then we have a `loadSprite` action, which creates a new `Image` object and loads the sprite. When the image is done loading, we set `this.sprite`.

Our `@computed` getters make it easier to access `MarbleDefinitions`. `marbleTypes` gives us a list of available types of marbles and `marbleDefinitions` returns the definitions object.

Running your code won't work just yet. We need the physics store first because it defines marble positions.

Part 2: Building the physics

Our whole physics engine fits into a single MobX store. It contains the collision detection, marble movement calculations, and drives the game loop itself.

The general approach goes like this:

1. Have an observable array of marbles
2. Run a `simulationStep` on each `requesAnimationFrame` using `d3.timer`
3. Change marble positions and speed

4. MobX observables and observers trigger re-renders of marbles that move

The whole Physics store[106] is some 120 lines of code, so I won't share it all at once. Here's what its skeleton looks like:

Physics skeleton

```
// src/logic/Physics.js

class Physics {
    @observable MarbleR = 25;
    @observable width = 800;
    @observable height = 600;
    @observable marbles = [];
    timer = null;

    @computed get initialPositions() {

    }

    @action startGameLoop() {

    }

    @action simulationStep() {

    }

    @action shoot({ x, y, vx, vy }, i) {

    }
}
```

We have four observable properties, a `timer`, a `@computed` property for initial positions, and 3 actions. The `startGameLoop` starts our game, `simulationStep` holds our main logic, and `shoot` shoots a particular marble.

Let's walk through.

initialPositions

[106] https://github.com/Swizec/declarative-canvas-react-konva/blob/master/src/logic/Physics.js

initialPositions function

```
// src/logic/Physics.js
class Physics {
                // ..
    @computed get initialPositions() {
        const { width, height, MarbleR } = this,
              center = width/2;

        const lines = 4,
              maxY = 200;

        let marbles = range(lines, 0, -1).map(y => {
            if (y === lines) return [{ x: center, y: maxY,
                                       vx: 0, vy: 0, r: this.MarbleR}];

            const left = center - y*(MarbleR+5),
                  right = center + y*(MarbleR+5);

            return range(left, right, MarbleR*2+5).map(x => ({
                x: x,
                y: maxY-y*(MarbleR*2+5),
                vx: 0,
                vy: 0,
                r: this.MarbleR
            }));
        }).reduce((acc, pos) => acc.concat(pos), []);

        marbles = [].concat(marbles, {
            x: width/2,
            y: height-150,
            vx: 0,
            vy: 0,
            r: this.MarbleR
        });

        marbles.forEach((m, i) => marbles[i].id = i);

        return marbles;
    }
    // ..
}
```

Believe it or not, this is like one of those *"Arrange things in a triangle"* puzzles you'd see in an old Learn How To Program book. Or a whiteboard interview.

It took me 3 hours to build. Super easy to get wrong and very fiddly to implement.

We start with a `range` of numbers. From `lines` to `0` in descending order. We iterate through this list of rows and change each into a list of marbles.

4 marbles in the first row, 3 in the next, all the way down to 1 in last row.

For each row, we calculate how much space we have on the `left` and `right` of the center and make a `range` of horizontal positions from `left` to `right` with a step of "1 marble size". Using these positions and the known row, we create marbles as needed.

We use a `.reduce` to flatten nested arrays and add the last marble. That's a corner case I couldn't solve elegantly, but I'm sure it's possible.

In the end, we add an `id` to each marble. We're using index as the id, that's true, but that still ensures we use consistent values throughout our app. Positions in the array may change.

shoot and startGameLoop

shoot and startGameLoop functions

```
// src/logic/Physics.js
class Physics {
    // ...

    @action startGameLoop() {
        this.marbles = this.initialPositions;

        this.timer = timer(() => this.simulationStep());
    }

    // ...

    @action shoot({ x, y, vx, vy }, i) {
        const maxSpeed = 20;

        this.marbles[i].x = x;
        this.marbles[i].y = y;
        this.marbles[i].vx = vx < maxSpeed ? vx : maxSpeed;
        this.marbles[i].vy = vy < maxSpeed ? vy : maxSpeed;
    }
}
```

`shoot` and `startGameLoop` are the simplest functions in our physics engine. `startGameLoop` creates the initial `marbles` array, and `shoot` updates a specific marble's coordinates and speed vector.

Lovely :smile:

simulationStep – where collisions collision

Here comes the fun part. The one with our game loop.

I've also made a video of all this. Watch it on YouTube[107]. It involves hand-drawn sketches that explain the math, and I think that's neat.

Full simulationStep function

```
@action simulationStep() {
    const { width, height, MarbleR } = this;

    const moveMarble = ({x, y, vx, vy, id}) => {
        let _vx = ((x+vx < MarbleR) ? -vx : (x+vx > width-MarbleR) ? -vx : vx)*.\
99,
            _vy = ((y+vy < MarbleR) ? -vy : (y+vy > height-MarbleR) ? -vy : vy)*\
.99;

        // nearest marble is a collision candidate
        const subdividedSpace = quadtree().extent([[-1, -1],
                                                   [this.width+1, this.height+1]\
])
                                          .x(d => d.x)
                                          .y(d => d.y)
                                          .addAll(this.marbles
                                                      .filter(m => id !== m.id)),
              candidate = subdividedSpace.find(x, y, MarbleR*2);

        if (candidate) {

            // borrowing @air_hadoken's implementation from here:
            // https://github.com/airhadoken/game_of_circles/blob/master/circles\
.js#L64
            const cx = candidate.x,
                  cy = candidate.y,
                  normx = cx - x,
                  normy = cy - y,
                  dist = (normx ** 2 + normy ** 2),
```

[107] https://www.youtube.com/watch?v=H84fmXjTElM

```
                c = (_vx * normx + _vy * normy) / dist * 2.3;

            _vx = (_vx - c * normx)/2.3;
            _vy = (_vy - c * normy)/2.3;

            candidate.vx += -_vx;
            candidate.vy += -_vy;
            candidate.x += -_vx;
            candidate.y += -_vy;
        }

        return {
            x: x + _vx,
            y: y + _vy,
            vx: _vx,
            vy: _vy
        }
    };

    this.marbles.forEach((marble, i) => {
        const { x, y, vx, vy } = moveMarble(marble);

        this.marbles[i].x = x;
        this.marbles[i].y = y;
        this.marbles[i].vx = vx;
        this.marbles[i].vy = vy;
    });
}
```

That's a lot of code 😅. Let's break it down.

You can think of `simulationStep` as a function and a loop. At the bottom, there is a `.forEach` that applies a `moveMarble` function to each marble.

Loop through marbles

```
    this.marbles.forEach((marble, i) => {
        const { x, y, vx, vy } = moveMarble(marble);

        this.marbles[i].x = x;
        this.marbles[i].y = y;
        this.marbles[i].vx = vx;
        this.marbles[i].vy = vy;
    });
```

We iterate over the list of marbles, feed them into `moveMarble`, get new properties, and save them in the main marbles array. This might be unnecessary because of MobX. We *should* be able to change them inside the loop and rely on MobX observables to do the heavy lifting.

I wonder why I did it like this :thinking_face: Maybe a leftover from before MobX…

moveMarble

`moveMarble` is itself a hairy function. Things happen in 3 steps:

1. Handle collisions with walls
2. Find collision with closest other marble
3. Handle collision with marble

Handling collisions with walls happens in two lines of code. One per coordinate.

Detecting wall collisions

```
let _vx = ((x+vx < MarbleR) ? -vx : (x+vx > width-MarbleR) ? -vx : vx)*.99,
    _vy = ((y+vy < MarbleR) ? -vy : (y+vy > height-MarbleR) ? -vy : vy)*.99;
```

Nested ternary expressions are kinda messy, but good enough. If a marble is beyond any boundary, we reverse its direction. We *always* apply a `.99` friction coefficient so that marbles slow down.

Finding collisions with the next closest marble happens via a quadtree. Since we don't have too many marbles, we can afford to build a new quadtree from scratch every time.

> A quadtree is a good way to subdivide space into areas. It lets us answer the question of "What's close enough to me to possibly touch me?" without making too many position comparisons.
>
> Checking every marble with every other marble produces 81 comparisons. Versus 2 comparisons using a quadtree.

Finding collision candidates

```
// nearest marble is a collision candidate
const subdividedSpace = quadtree().extent([[-1, -1],
                                           [this.width+1, this.height+1]])
                                  .x(d => d.x)
                                  .y(d => d.y)
                                  .addAll(this.marbles
                                              .filter(m => id !== m.id)),
      candidate = subdividedSpace.find(x, y, MarbleR*2);
```

We're using `d3-quadtree`[108] for the quadtree implementation. It takes an `extent`, which tells it how big our space is. It uses `x` and `y` accessors to get coordinates out of our marble objects, and we use `addAll` to fill it with marbles.

To avoid detecting each marble as colliding with itself, we take each marble out of our list before feeding the quadtree.

Once we have a quadtree built out, we use `.find` to look for the nearest marble within `MarbleR*2` of the current marble. Which is exactly the one we're colliding with! :smile:

Handling collisions with marbles involves math. The sort of thing you think you remember from high school, and suddenly realize you don't when the time comes to use it.

The code looks like this:

Handling marble collisions

```
if (candidate) {

    // borrowing @air_hadoken's implementation from here:
    // https://github.com/airhadoken/game_of_circles/blob/master/circles.js#L64
    const cx = candidate.x,
          cy = candidate.y,
          normx = cx - x,
          normy = cy - y,
          dist = (normx ** 2 + normy ** 2),
          c = (_vx * normx + _vy * normy) / dist * 2.3;

    _vx = (_vx - c * normx)/2.3;
    _vy = (_vy - c * normy)/2.3;

```

[108] https://github.com/d3/d3-quadtree

```
    candidate.vx += -_vx;
    candidate.vy += -_vy;
    candidate.x += -_vx;
    candidate.y += -_vy;
}

return {
    x: x + _vx,
    y: y + _vy,
    vx: _vx,
    vy: _vy
}
```

Ok, the `return` statement isn't about handling collisions. It updates the current marble.

The rest kind of looks like magic. I implemented it, and it looks like magic, and I feel like I don't *really* understand it.

You can think of `[normx, normy]` as a vector that points from current marble to collision candidate. It gives us bounce direction. We use the euclidean distance[109] formula to calculate the length of this vector. The distance between the centers of both marbles.

Then we calculate the dot product[110] between our marble's speed vector and the collision direction vector. And we normalize it by distance. Multiplying distance by `2` accounts for there being two marbles in the collision. That extra `.3` made the simulation look better.

I fiddled with it :smile:

Then we use the dot product scalar to adjust the marble's speed vector. Dividing by `2` takes into account that half the energy goes to the other marble. This is only true because we assume their masses are equal.

Finally, we update the `candidate` marble and make sure it bounces off as well. We do it additively because that's how it happens in real life.

Now two marbles traveling towards each other in exactly opposite directions with exactly the same speed, will stop dead and stay there. As soon as there's any misalignment, deflection happens. If one is stationary, it starts moving. If it's moving in the same direction, it speeds up... etc.

The end result is a decent-looking simulation of billiards[111]. I'd show you a gif, but those don't work well in books.

[109] https://en.wikipedia.org/wiki/Euclidean_distance
[110] https://en.wikipedia.org/wiki/Dot_product
[111] https://swizec.github.io/declarative-canvas-react-konva/

Using a React alternative like Preact or Inferno

We've been using React so far, and that's worked great for us. React is fast, easy to use, and not that hard to understand. However, two up-and-coming frameworks promise the ease of React, but faster.

Preact[112] looks just like React when you're using it, but the smaller code footprint means your apps open faster. Performing fewer sanity checks at runtime also makes it faster to run. Async rendering has the potential to make your apps feel faster as well.

Inferno[113] also promises to look just like React when you're using it, but its sole purpose in life is to be faster. According to its maintainers, converting to Inferno can improve performance of your real-world app up to 110%.

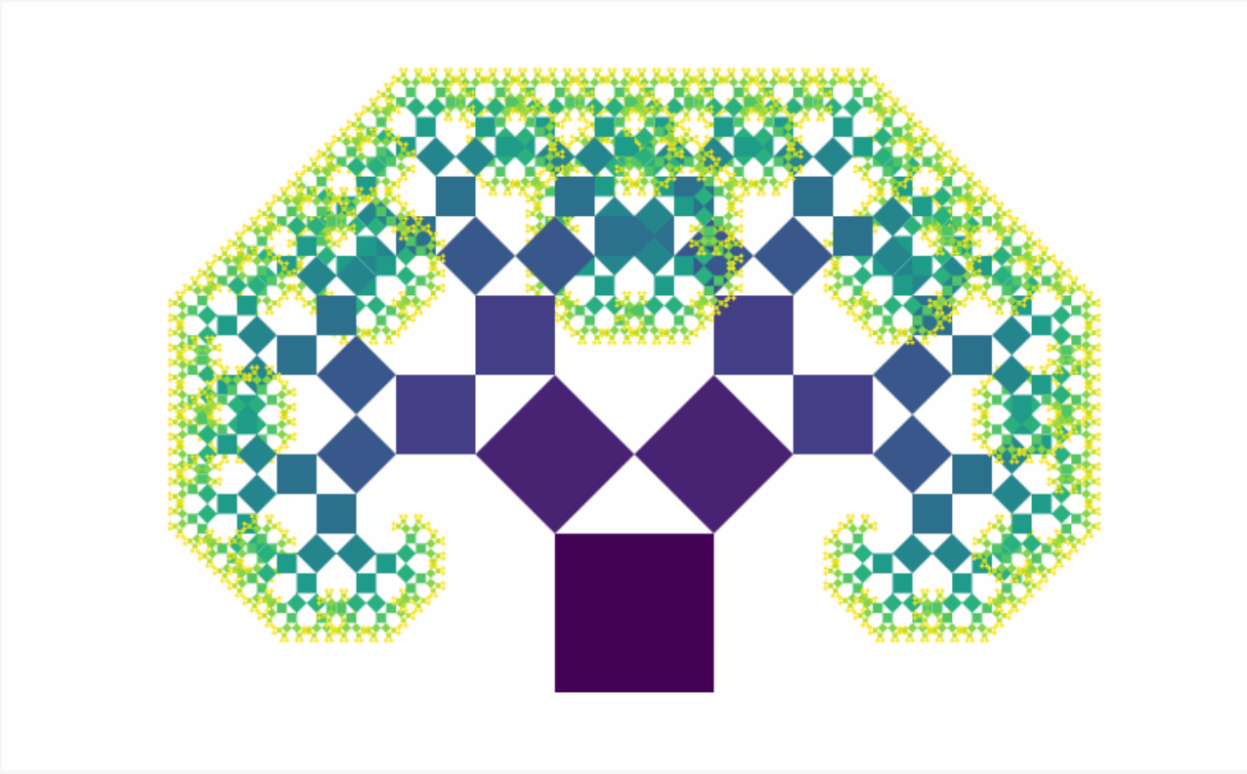
Both Preact and Inferno have a `-compat` project that lets you convert existing React projects without any code modifications. Whether they remain faster than React after the new React Fiber updates remains to be seen.

As of this writing, July 2017, React Fiber is in alpha. The team is promising it's going to be faster than current React using many of the same tricks that Preact and Inferno use. The big update everyone is looking for is the new render scheduler that's going to make things feel faster even when they aren't. We'll see.

Stress testing with a recursive fractal

To show you how these speed improvements look in real life, I've devised a stress test. A pythagorean fractal tree[114] that moves around when you move your mouse.

[112] https://github.com/developit/preact
[113] https://github.com/infernojs/inferno
[114] https://en.wikipedia.org/wiki/Pythagoras_tree_(fractal)



Pythagorean tree

It's a great stress test because a render like this is the worst case scenario for tree-based rendering. You have 2048 SVG nodes, deeply nested, that all change props and re-render with every change.

You can see the full code on Github[115]. We're going to focus on the recursive `<Pythagoras>` component here.

How you too can build a dancing tree fractal

Equipped with basic trigonometry, you need 3 ingredients to build a dancing tree:

- a recursive `<Pythagoras>` component
- a mousemove listener
- a memoized next-step-props calculation function

We're using a `<Pythagoras>` component for each square and its two children, a D3 mouse listener, and some math that a reader helped me with. We need D3 because its mouse listeners automatically calculate mouse position relative to SVG coordinates. Memoization[116] in the math function helps us keep our code faster.

The `<Pythagoras>` component looks like this:

[115] https://github.com/Swizec/react-fractals
[116] https://en.wikipedia.org/wiki/Memoization

Recursive <Pythagoras> component

```
const Pythagoras = ({ w, x, y, heightFactor, lean, left, right, lvl, maxlvl }) =\
> {
    if (lvl >= maxlvl || w < 1) {
        return null;
    }

    const { nextRight, nextLeft, A, B } = memoizedCalc({
        w: w,
        heightFactor: heightFactor,
        lean: lean
    });

    let rotate = '';

    if (left) {
        rotate = `rotate(${-A} 0 ${w})`;
    }else if (right) {
        rotate = `rotate(${B} ${w} ${w})`;
    }

    return (
        <g transform={`translate(${x} ${y}) ${rotate}`}>
            <rect width={w} height={w}
                  x={0} y={0}
                  style={{fill: interpolateViridis(lvl/maxlvl)}} />

            <Pythagoras w={nextLeft}
                        x={0} y={-nextLeft}
                        lvl={lvl+1} maxlvl={maxlvl}
                        heightFactor={heightFactor}
                        lean={lean}
                        left />

            <Pythagoras w={nextRight}
                        x={w-nextRight} y={-nextRight}
                        lvl={lvl+1} maxlvl={maxlvl}
                        heightFactor={heightFactor}
                        lean={lean}
                        right />

        </g>
```

```
    );
};
```

We break recursion whenever we try to draw an invisible square or have reached too deep into the tree. Then we:
- use `memoizedCalc` to do the mathematics
- define different `rotate()` transforms for the `left` and `right` branches
- and return an SVG `<rect>` for the current rectangle, and two `<Pythagoras>` elements for each branch.

Most of this code deals with passing arguments onwards to children, which isn't the most elegant approach, but it works. The rest is about positioning branches so corners match up.

Corners matching up

The math

I don't *really* understand this math, but I sort of know where it's coming from. It's the sine law[117] applied correctly. The part I failed at miserably when I tried[118].

Trigonometry that moves our fractal

```
const memoizedCalc = function () {
    const memo = {};

    const key = ({ w, heightFactor, lean }) => [w,heightFactor, lean].join('-');

    return (args) => {
        const memoKey = key(args);

        if (memo[memoKey]) {
            return memo[memoKey];
        }else{
            const { w, heightFactor, lean } = args;
```

[117] https://en.wikipedia.org/wiki/Law_of_sines
[118] https://swizec.com/blog/fractals-react/swizec/7233

```
13
14              const trigH = heightFactor*w;
15
16              const result = {
17                  nextRight: Math.sqrt(trigH**2 + (w * (.5+lean))**2),
18                  nextLeft: Math.sqrt(trigH**2 + (w * (.5-lean))**2),
19                  A: Math.deg(Math.atan(trigH / ((.5-lean) * w))),
20                  B: Math.deg(Math.atan(trigH / ((.5+lean) * w)))
21              };
22
23              memo[memoKey] = result;
24              return result;
25          }
26      }
27  }();
```

We expanded the basic law of sines with a dynamic `heightFactor` and `lean` adjustment. We'll control those with mouse movement.

To improve performance, maybe, our `memoizedCalc` function has an internal data store that maintains a hash of every argument tuple and its result. This lets us avoid computation and read from memory instead.

At 11 levels of recursion, `memoizedCalc` is called 2,048 times and only returns 11 different results. You can't find a better candidate for memoization.

Of course, a benchmark would be great here. Maybe `sqrt`, `atan`, and `**` aren't *that* slow, and our real bottleneck is redrawing all those nodes on every mouse move. Hint: it totally is.

Our ability to memoize calculation also means that we could, in theory, flatten our rendering. Instead of recursion, we could use iteration and render in layers. Squares at each level are the same, just at different `(x, y)` coordinates.

This, however, would be tricky to implement. A lot more computation, I think.

The mouse listener

Inside `App.js`[119], we add a mouse event listener. We use D3's because it gives us SVG-relative position calculation out of the box. With React's, we'd have to do the hard work ourselves.

[119] https://github.com/Swizec/react-fractals/blob/master/src/App.js

Reacting to mouse movement

```
// App.js
state = {
        currentMax: 0,
        baseW: 80,
        heightFactor: 0,
        lean: 0
    };

componentDidMount() {
        d3select(this.refs.svg)
           .on("mousemove", this.onMouseMove.bind(this));
}

onMouseMove(event) {
        const [x, y] = d3mouse(this.refs.svg),

        scaleFactor = scaleLinear().domain([this.svg.height, 0])


        scaleLean = scaleLinear().domain([0, this.svg.width/2, this.svg.width])


        this.setState({
                heightFactor: scaleFactor(y),
                lean: scaleLean(x)
        });
}

// ...

render() {
        // ...
        <svg ref="svg"> //...
        <Pythagoras w={this.state.baseW}
                                                h={this.state.baseW}
                                                heightFactor={this.state.heightFactor}
                                        lean={this.state.lean}
                                        x={this.svg.width/2-40}
                                        y={this.svg.height-this.state.baseW}
                                        lvl={0}
                                        maxlvl={this.state.currentMax}/>
```

```
42  }
```

A few things happen here:

- we set initial `lean` and `heightFactor` to `0`
- in `componentDidMount`, we use `d3.select` and `.on` to add a mouse listener
- we define an `onMouseMove` method as the listener
- render the first `<Pythagoras>` using values from `state`

The `lean` parameter tells us which way the tree is leaning and by how much, the `heightFactor` tells us how high those triangles should be. We control both with the mouse position.

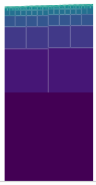
That happens in `onMouseMove`:

Moving the fractal as user moves mouse

```
1   onMouseMove(event) {
2           const [x, y] = d3mouse(this.refs.svg),
3
4           scaleFactor = scaleLinear().domain([this.svg.height, 0])
5
6
7           scaleLean = scaleLinear().domain([0, this.svg.width/2, this.svg.width])
8
9
10          this.setState({
11                  heightFactor: scaleFactor(y),
12                  lean: scaleLean(x)
13          });
14  }
```

`d3mouse` - an imported `mouse` function from `d3-selection` - gives us cursor position relative to the SVG element. Two linear scales give us `scaleFactor` and `scaleLean` values, which we put into component state.

When we feed a change to `this.setState`, it triggers a re-render of the entire tree, our `memoizedCalc` function returns new values, and the final result is a dancing tree.



Dancing Pythagoras tree

Beautious. :heart_eyes:

But since this is a book, you probably can't see the gif move. I suggest looking at this helpful version[120] on Github.

Trying the stress test in Preact and Inferno

The Pythagoras tree example got converted into 6 different UI libraries: React, Preact, Inferno, Vue, Angular 2, and CycleJS.

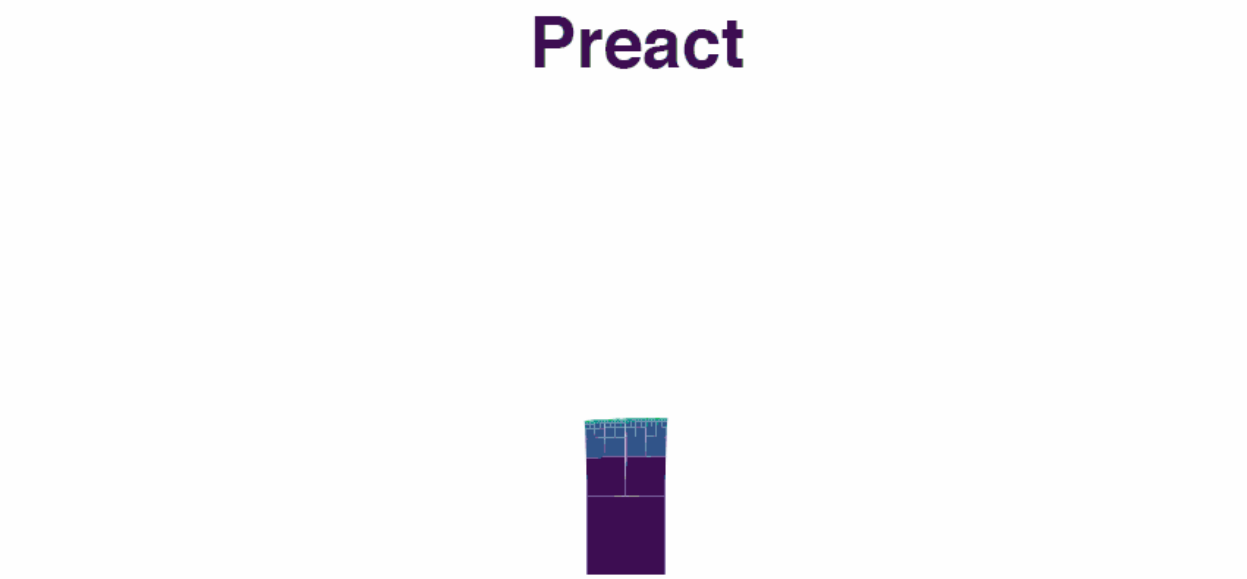
Mostly by the creators of those libraries themselves because they're awesome and much smarter than I am. You can see the result, in gifs, on my blog here[121].

We're focusing on React-like libraries in this book, so here's a rundown of changes required for Preact and Inferno with links to full code.

Preact

Implemented by Jason Miller, creator of Preact. Full code on Github[122]

[120] https://github.com/Swizec/react-fractals/blob/master/react-tree.gif
[121] https://swizec.com/blog/animating-svg-nodes-react-preact-inferno-vue/swizec/7311
[122] https://github.com/developit/preact-fractals



Jason used the `preact-compat` layer to make Preact pretend that it's React. This might impact performance.

What I love about the Preact example is that it uses async rendering to look smoother. You can see the redraw cycle lag behind the mouse movement producing curious effects.

I like it.

Here's how he did it: full diff on github[123]

In `package.json`, he added `preact`, `preact-compat`, and preact -compat clones for React libraries. I guess you need the latter so you don't have to change your imports.

He changed the functional stateless `Pythagoras` component into a stateful component to enable async rendering.

Change <Pythagoras> to a class

```
1 // src/Pythagoras.js
2 export default class {
3         render(props) {
4                 return Pythagoras(props);
5         }
6 }
```

And enabled debounced asynchronous rendering:

[123] https://github.com/Swizec/react-fractals/compare/master...developit:master

Debounce rendering

```
// src/index.js
import { options } from 'preact';
options.syncComponentUpdates = false;

//option 1:  rIC + setTimeout fallback
let timer;
options.debounceRendering = f => {
    clearTimeout(timer);
    timer = setTimeout(f, 100);
    requestIdleCallback(f);
};
```

My favorite part is that you can use Preact as a drop-in replacement for React and it Just Works™ *and* works well. This is promising for anyone who wants a quick win in their codebase.

Inferno

Implemented by Dominic Gannaway, creator of Inferno. Full code on Github[124]

You *can* use Inferno as a drop-in replacement for React, and I did when I tried to build this myself. Dominic says that impacts performance, though, so he made a proper fork. You can see the full diff on Github[125]

Dominic changed all `react-scripts` references to `inferno-scripts`, and it's a good sign that such a thing exists. He also changed the `react` dependency to `inferno-beta36`, but I'm sure it's out of beta by the time you're reading this. The experiment was done in December 2016.

From there, the main changes are to the imports – React becomes Inferno – and he changed some class methods to bound fat arrow functions. I don't know if that's a stylistic choice or an Inferno requirement.

He also had to change a string-based ref into a callback ref. Inferno doesn't do string-based refs for performance reasons, and we need refs so we can use D3 to detect mouse position on SVG.

[124] https://github.com/trueadm/inferno-fractals
[125] https://github.com/Swizec/react-fractals/compare/master...trueadm:master

Change ref to callback

```
// src/App.js

class App extends Component {
        // ...
        svgElemeRef = (domNode) => {
                this.svgElement = domNode;
        }
        // ...
        render() {
                // ..
                <svg width={this.svg.width} height={this.svg.height} ref={this.svgElemeRef }>
        }
```

In the core `Pythagoras` component, he added two Inferno-specific props: `noNormalize` and `hasNonKeyedChildren`.

According to this issue[126], `noNormalize` is a benchmark-focused flag that improves performance, and I can't figure out what `hasNonKeyedChildren` does. I assume both are performance optimizations for the Virtual DOM diffing algorithm.

[126] https://github.com/trueadm/inferno/issues/565

Conclusion

You finished the book! You're my new favorite person \o/

You've built a really big interactive data visualization, a few silly animations, learned the basics of Redux and MobX, looked into rendering your stuff on canvas, and considered React alternatives.

Right now you might be thinking *"How the hell does this all fit together to help me build interactive graphs and charts?"* You've learned the building blocks!

First, you learned the basics. How to make React and D3 like each other, how to approach rendering, what to do with state. Then you built a big project to see how it fits together.

Then you learned animation basics. How to approach updating components with fine grained control and how to do transitions. You learned that tricks from the gaming industry work here, too.

You're an expert React slinger now! I think that's awesome.

Even if you didn't build the projects, you still learned the theory. When the right problem presents itself, you'll think of this book and remember what to do. Or at least know where to look to find a solution.

At the very least, you know where to look to find a solution. This book is yours forever.

I hope you have enjoyed this book. Tweet @Swizec[127] and tell me what you're building. :smile:

Questions? Poke me on twitter. I'm @Swizec[128]. Or send me an email to hi@swizec.com. I read every email and reply on a best effort basis.

[127] https://twitter.com/Swizec
[128] https://twitter.com/Swizec

Appendixes

Appendix A - roll your own environment

If you already know how to set up the perfect development environment for modern JavaScript, go ahead and skip this section. Otherwise, keep reading.

If you don't know and you don't care about this right now: Use the starter project that came with the book. It's what you would get after following this chapter.

A good work environment helps us get the most out of our time. We're after three things:

- code should re-compile when we change a file
- page should update automatically when the code changes
- dependencies and modules should be simple to manage

When I first wrote this chapter in April 2015, I suggested a combination of Browserify, Grunt, NPM, and Bower. This was the wrong approach. It was complicated, it wasn't very extensible, and it was slow.

There were no sourcemaps, which meant your browser's error reporting was useless. There was no hot loading, which meant your code had to process a gigantic 80,000 datapoint file every time you made a change.

It was a mess. I'm sorry I told you to use it. The old system is included in the appendix if you're curious.

The new system is great. I promise. I use it all the time :)

Bundle with Webpack

Instead of using the old setup, I now think the best choice is to use a combination of Webpack and Babel.

Webpack calls itself a *"flexible unbiased extensible module bundler"*, which sounds like buzzword soup. At its most basic, Webpack gives you the ability to organize code into modules and `require()` what you need, much like Browserify.

Unlike Browserify however, Webpack comes with a sea of built-in features and a rich ecosystem of extensions called plugins. I can't hope to know even half of them, but some of the coolest I've used are plugins that let you `require()` Less files *with* magical Less-to-CSS compilation, plugins for `require`-ing images, and JavaScript minification.

Using Webpack allows us to solve two more annoyances — losing state when loading new code and accurately reporting errors. So we can add two more requirements to our work environment checklist (for a total of five):

- code should re-compile when we change a file
- page should update automatically when the code changes
- dependencies and modules should be simple to manage
- page shouldn't lose state when loading new code (hot loading)
- browser should report errors accurately in the right source files (sourcemaps)

Compile with Babel

Webpack can't do all this alone though – it needs a compiler.

We're going to use Babel to compile our JSX and ES6 code into the kind of code all browsers understand: ES5. If you're not ready to learn ES6, don't worry; you can read the ES5 version of React+d3.js.

Babel isn't really a compiler because it produces JavaScript, not machine code. That being said, it's still important at this point. According to the JavaScript roadmap, browsers aren't expected to fully support ES6 until some time in 2017. That's a long time to wait, so the community came up with transpilers which let us use some ES6 features *right now*. Yay!

Transpilers are the officially-encouraged stepping stone towards full ES6 support.

To give you a rough idea of what Babel does with your code, here's a fat arrow function with JSX. Don't worry if you don't understand this code yet; we'll go through that later.

```
() ⇒ (<div>Hello there</div>)
```

After Babel transpiles that line into ES5, it looks like this:

```
(function () {
    return React.createElement(
      'div',
      null,
      'Hello there'
    );
  });
```

Babel developers have created a playground that live-compiles[129] code for you. Have fun.

[129] https://babeljs.io/repl/

Quickstart

The quickest way to set this up is to use Dan Abramov's react-transform-boilerplate[130] project. It's what I use for new projects these days.

If you know how to do this already, skip ahead to the Visualizing Data with React.js chapter. In the rest of this chapter, I'm going to show you how to get started with the boilerplate project. I'm also going to explain some of the moving parts that make it tick.

Your book package also contains a starter project. You can use that to get started right away. It's what you would get after following this chapter.

NPM for dependencies and tools

NPM is node.js's default package manager. Originally developed as a dependency management tool for node.js projects, it's since taken hold of the JavaScript world as a way to manage the toolbelt. With Webpack's growing popularity and its ability to recognize NPM modules, NPM is fast becoming *the* way to manage client-side dependencies as well.

We're going to use NPM to install both our toolbelt dependencies (like Webpack) and our client-side dependencies (like React and d3.js).

You can get NPM by installing node.js from nodejs.org[131]. Webpack and our dev server will run in node.

Once you have NPM, you can install Webpack globally with:

```
$ npm install webpack -g
```

If that worked, you're ready to go. If it didn't, Google is your friend.

At this point, there are two ways to proceed:

- You can continue with the step-by-step instructions using a boilerplate.
- You can use the stub project included with the book. It has everything ready to go.

Step-by-step with boilerplate

All it takes to start a project from boilerplate is to clone the boilerplate project, remove a few files, and run the code to make sure everything works.

[130] https://github.com/gaearon/react-transform-boilerplate
[131] http://nodejs.org

You will need Git for this step. I assume you have it already because you're a programmer. If you don't, you can get it from Git's homepage[132]. For the uninitiated, Git is a source code versioning tool.

Head to a directory of your choosing, and run:

$ git clone git@github.com:gaearon/react-transform-boilerplate.git

This makes a local copy of the boilerplate project. Now that we've got the base code, we should make it our own.

Our first step is to rename the directory and remove Git's version history and ties to the original project. This will let us turn it into our own project.

$ mv react-transform-boilerplate react-d3-example
$ rm -rf react-d3-example/.git
$ cd react-d3-example

We now have a directory called `react-d3-example` that contains some config files and a bit of code. Most importantly, it isn't tied to a Git project, so we can make it all ours.

Make it your own

To make the project our own, we have to change some information inside `package.json`: the name, version, and description.

```
// ./package.json
 {
  // leanpub-start-delete
   "name": "react-transform-boilerplate",
   "version": "1.0.0",
    "description": "A new Webpack boilerplate with hot reloading React components, and error handling on module and component level.",
  // leanpub-end-delete
  // leanpub-start-insert
   "name": "react-d3-example",
   "version": "0.1.0",
   "description": "An example project to show off React and d3 working together",
  // leanpub-end-insert
   "scripts": {
```

It's also a good idea to update the `author` field:

[132] https://git-scm.com/

```
// ./src/package.json
  // leanpub-start-delete
    "author": "Dan Abramov dan.abramov@me.com (http://github.com/gaearon)",
  // leanpub-end-delete
  // leanpub-start-insert
  "author": "Swizec swizec@swizec.com (http://swizec.com)
  // leanpub-end-insert
```

Use your own name, email, and URL. Not mine :)

If you want to use Git to manage source code versioning, now is a good time to start. You can do that by running:

```
$ git init
    $ git add .
    $ git commit -a -m "Start project from boilerplate"
```

Great. Now we have a project signed in our name.

Our new project comes preconfigured for React and all the other tools and compilers we need to run our code. Install them by running:

```
$ npm install
```

This installs a bunch of dependencies like React, a few Webpack extensions, and a JavaScript transpiler (Babel) with a few bells and whistles. Sometimes, parts of the installation fail. If it happens to you, try re-running `npm install` for the libraries that threw an error. I don't know why this happens, but you're not alone. I've been seeing this behavior for years.

Now that we have all the basic libraries and tools, we have to install two more libraries:

1. `d3` for drawing
2. `lodash` for some utility functions

```
$ npm install –save d3 lodash
```

The `--save` option saves them to `package.json`.

Add Less compiling

Less is my favorite way to write stylesheets. It looks almost like traditional CSS, but it gives you the ability to use variables, nest definitions, and write mixins. We won't need much of this for the H1B

graphs project, but nesting will make our style definitions nicer, and Less makes your life easier in bigger projects.

Yes, there's a raging debate about stylesheets versus inline styling. I like stylesheets for now.

Webpack can handle compiling Less to CSS for us. We need a couple of Webpack loaders and add three lines to the config.

Let's start with the loaders:

```
$ npm install –save style-loader less less-loader css-loader
```

Remember, `--save` adds `style-loader` and `less-loader` to package.json. The `style-loader` takes care of transforming `require()` calls into `<link rel="stylesheet"` definitions, and `less-loader` takes care of compiling LESS into CSS.

To add them to our build step, we have to go into `webpack.config.dev.js`, find the `loaders: [` definition, and add a new object. Like this:

<<Add LESS loaders[133]

Don't worry if you don't understand what the rest of this file does. We're going to look at that in the next section.

Our addition tells Webpack to load any files that end with `.less` using `style!css!less`. The `test:` part is a regex that describes which files to match, and the `loader` part uses bangs to chain three loaders. The file is first compiled with `less`, then compiled into `css`, and finally loaded as a `style`.

If everything went well, we should now be able to use `require('./style.less')` to load style definitions. This is great because it allows us to have separate style files for each component, and that makes our code more reusable since every module comes with its own styles.

Serve static files in development

Our visualization is going to use Ajax to load data. That means the server we use in development can't just route everything to `index.html` – it needs to serve other static files as well.

We have to add a line to `devServer.js`:

<<Enable static server on ./public[134]

This tells express.js, which is the framework our simple server uses, to route any URL starting with `/public` to local files by matching paths.

[133] code_samples/env/webpack.config.dev.js
[134] code_samples/env/devServer.js

Webpack nice-to-haves

Whenever I'm working with React, I like to add two nice-to-haves to `webpack.config.dev.js`. They're not super important, but they make my life a little easier.

First, I add the `.jsx` extension to the list of files loaded with Babel. This lets me write React code in `.jsx` files. I know what you're thinking: writing files like that is no longer encouraged by the community, but hey, it makes my Emacs behave better.

<<Add .jsx to Babel file extensions[135]

We changed the `test` regex to add `.jsx`. You can read in more detail about how these configs work in later parts of this chapter.

Second, I like to add a `resolve` config to Webpack. This lets me load files without writing their extension. It's a small detail, but it makes your code cleaner.

<<Add resolve to webpack.config.dev.js[136]

It's a list of file extensions that Webpack tries to guess when a path you use doesn't match any files.

Optionally enable ES7

Examples in this book are written in ES6, also known as ECMAScript2015. If you're using the boilerplate approach or the stub project you got with the book, all ES6 features work in any browser. The Babel 6 compiler makes sure of that by transpiling ES6 into ES5.

The gap between ES5 (2009/2011) and ES6 (2015) has been long, but now the standard has started moving fast once again. In fact, ES7 is promised to be released some time in 2016.

Even though ES7 hasn't been standardized yet, you can already use some of its features if you enable `stage-0` support in Babel. Everything in `stage-0` is stable enough to use in production, but there is a small chance the feature/syntax will change when the new standard becomes official.

You don't need `stage-0` to follow the examples in this book, but I do use one or two syntax sugar features. Whenever we use something from ES7, I will mention the ES6 alternative.

To enable `stage-0`, you have to first install `babel-preset-stage-0`. Like this:

```
$ npm install –save-dev babel-preset-stage-0
```

Then enable it in `.babelrc`:

<<Add stage-0 preset to .babelrc[137]

[135] code_samples/env/webpack.config.dev.js
[136] code_samples/env/webpack.config.dev.js
[137] code_samples/env/babelrc

That's it. You can use fancy ES7 features in your code and Babel will transpile them into normal ES5 that all browsers support.

Don't worry if you don't understand how `.babelrc` works. You can read more about the environment in the Environment in depth chapter.

Check that everything works

Your environment should be ready. Let's try it out. First, start the dev server:

```
$ npm start
```

This command runs a small static file server that's written in node.js. The server ensures that Webpack continues to compile your code when it detects a file change. It also puts some magic in place that hot loads code into the browser without refreshing and without losing variable values.

Assuming there were no errors, you can go to `http://localhost:3000` and see a counter doing some counting. That's the sample code that comes with the boilerplate.

If it worked: Awesome, you got the code running! Your development environment works! Hurray!

If it didn't work: Well, um, poop. A number of things could have gone wrong. I would suggest making sure `npm install` ran without issue, and if it didn't, try Googling for any error messages that you get.

Remove sample code

Now that we know our development environment works, we can get rid of the sample code inside `src/`. We're going to put our own code files in there.

We're left with a skeleton project that's full of configuration files, a dev server, and an empty `index.html`. This is a good opportunity for another `git commit`.

Done? Wonderful.

In the rest of this chapter, we're going to take a deeper look into all the config files that came with our boilerplate. If you don't care about that right now, you can jump straight to the meat.

The environment in depth

Boilerplate is great because it lets you get started right away. No setup, no fuss, just `npm install` and away we go.

But you *will* have to change something eventually, and when you do, you'll want to know what to look for. There's no need to know every detail in every config file, but you do have to know enough so that you can Google for help.

Let's take a deeper look at the config files to make any future Googling easier. We're relying on Dan Abramov's `react-transform-boilerplate`, but many others exist with different levels of bells and whistles. I like Dan's because it's simple.

All modern boilerplates are going to include at least two bits:

- the webpack config
- the dev server

Everything else is optional.

Webpack config

Webpack is where the real magic happens, so this is the most important configuration file in your project. It's just a JavaScript file though, so there's nothing to fear.

Most projects have two versions of this file: a development version and a production version. The first is geared more towards what we need in development – a compile step that leaves our JavaScript easy to debug – while the second is geared towards what we need in production – compressed and uglified JavaScript that's quick to load.

Both files look alike, so we're going to focus on the dev version.

It comes in four parts:

<<Webpack config structure[138]

- **Entry**, which tells Webpack where to start building our project's dependency tree;

- **Output**, which tells Webpack where to put the result. This is what our index.html file loads;

- **Plugins**, which tells Webpack which plugins to use when building our code;

- and **Loaders**, which tells Webpack about the different file loaders we'd like to use.

There's also the `devtool: 'eval'` option, which tells Webpack how to package our files so they're easier to debug. In this case, our code will come inside `eval()` statements, which makes it hot loadable.

Let's go through the four sections one by one.

[138] code_samples/env/webpack.config.dev.js

Entry

The entry section of Webpack's config specifies the entry points of our dependency tree. It's an array of files that `require()` all other files.

In our case, it looks like this:

<<Entry part of webpack.config.dev.js[139]

We specify that `./src/index` is the main file. In the next section, you'll see that this is the file that requires our app and renders it into the page.

The `webpack-hot-middleware/client` line enables Webpack's hot loading. It loads new versions of JavaScript files without reloading the page.

Output

The output section specifies which files get the output. Our config is going to put all compiled code into a single `bundle.js` file, but it's common to have multiple output files. If we had an admin dashboard and a user-facing app, this would allow us to avoid loading unnecessary JavaScript for users who don't need every type of functionality.

The config looks like this:

<<Output part of webpack.config.dev.js[140]

We define a path, `./dist/`, where compiled files live, say the filename for JavaScript is `bundle.js`, and specify `/static/` as the public path. That means the `<script>` tag in our HTML should use `/static/bundle.js` to get our code, but we should use `./dist/bundle.js` to copy the compiled file.

Plugins

There's a plethora of Webpack plugins out there. We're only going to use two of them in our example.

<<Plugins part of webpack.config.dev.js[141]

As you might have guessed, this config is just an array of plugin object instances. Both plugins we're using come with Webpack by default. Otherwise, we'd have to `require()` them at the top of the file.

`HotModuleReplacementPlugin` is where the hot loading magic happens. I have no idea how it works, but it's the most magical thing that's ever happened to my coding abilities.

The `NoErrorsPlugin` makes sure that Webpack doesn't error out and die when there's a problem with our code. The internet recommends using it when you rely on hot loading new code.

[139] code_samples/env/webpack.config.dev.js
[140] code_samples/env/webpack.config.dev.js
[141] code_samples/env/webpack.config.dev.js

Loaders

Finally, we come to the loaders section. Much like with plugins, there is a universe of Webpack loaders out there, and I've barely scratched the surface.

If you can think of it, there's a loader for it. At my day job, we use a Webpack loader for everything from JavaScript code to images and font files.

For the purposes of this book, we don't need anything that fancy. We just need a loader for JavaScript and styles.

<<Loaders part of webpack.config.dev.js[142]

Each of these definitions comes in three parts:

- **`test`**, which specifies the regex for matching files;
- **`loader` or `loaders`**, which specifies which loader to use for these files. You can compose loader sequences with bangs, `!`;
- optional **`include`**, which specifies the directory to search for files.

There might be loaders out there with more options, but this is the most basic loader I've seen that covers our bases.

That's it for our very basic Webpack config. You can read about all the other options in Webpack's own documentation[143].

My friend Juho Vepsäläinen has also written a marvelous book that dives deeper into Webpack. You can find it at survivejs.com[144].

Dev server

The dev server that comes with Dan's boilerplate is based on the Express framework. It's one of the most popular frameworks for building websites in node.js.

Many better and more in-depth books have been written about node.js and its frameworks. In this book, we're only going to take a quick look at some of the key parts.

For example, on line 9, you can see that we tell the server to use Webpack as a middleware. That means the server passes every request through Webpack and lets it change anything it needs.

<<Lines that tell Express to use Webpack[145]

The `compiler` variable is an instance of Webpack, and `config` is the config we looked at earlier. `app` is an instance of the Express server.

[142] code_samples/env/webpack.config.dev.js
[143] http://webpack.github.io/docs/
[144] http://survivejs.com
[145] code_samples/env/devServer.js

Another important bit of the `devServer.js` file specifies routes. In our case, we want to serve everything from `public` as a static file, and anything else to serve `index.html` and let JavaScript handle routing.

<<Lines that tell Express how to route requests[146]

This tells Express to use a static file server for everything in `public` and to serve `index.html` for anything else.

At the bottom, there is a line that starts the server:

<<Line that starts the server[147]

I know I didn't explain much, but that's as deep as we can go at this point. You can read more about node.js servers, and Express in particular, in Azat Mardan's books[148]. They're great.

Babel config

Babel works great out of the box. There's no need to configure anything if you just want to get started and don't care about optimizing the compilation process.

But there are a bunch of configuration options[149] if you want to play around. You can configure everything from enabling and disabling ES6 features to source maps and basic code compacting and more. More importantly, you can define custom transforms for your code.

We don't need anything fancy for the purposes of our example project – just a few presets. A preset is a single package that enables a bunch of plugins and code transforms. We use them to make our lives easier, but you can drop into a more specific config if you want to.

The best way to configure Babel is through the `.babelrc` file, which looks like this:

<<.babelrc config[150]

I imagine this file is something most people copy-paste from the internet, but here's what's happening in our case:

- `react` enables all React and JSX plugins;
- `es2015` enables transpiling ES6 into ES5, including all polyfills for semantic features;
- `stage-0` enables the more experimental ES7 features.

Those are the default presets. For development, we also enable `react-hmre`, which gives us hot loading.

That's it. If you need more granular config, or you want to know what all those presets enable and use, I suggest Googling for them. Be warned, though; the `es2015` preset alone uses 20 different plugins.

[146] code_samples/env/devServer.js
[147] code_samples/env/devServer.js
[148] http://azat.co/
[149] http://babeljs.io/docs/usage/options/
[150] code_samples/env/babelrc

Editor config

A great deal has been written about tabs vs. spaces. It's one of the endless debates we programmers like to have. *Obviously* single quotes are better than double quotes… unless… well… it depends, really.

I've been coding since I was a kid, and there's still no consensus. Most people wing it. Even nowadays when editors come with a built-in linter, people still wing it.

But in recent months (years?), a solution has appeared: the `.eslintrc` file. It lets you define project-specific code styles that are programmatically enforced by your editor.

From what I've heard, most modern editors support `.eslintrc` out of the box, so all you have to do is include the file in your project. Do that and your editor keeps encouraging you to write beautiful code.

The `eslint` config that comes with Dan's boilerplate loads a React linter plugin and defines a few React-specific rules. It also enables JSX linting and modern ES6 modules stuff. By the looks of it, Dan is a fan of single quotes.

<<.eslintrc for React code[151]

I haven't really had a chance to play around with linting configs like these to be honest. Emacs defaults have been good to me for years. But these types of configs are a great idea. The biggest problem in a team is syncing everyone's linters, and if you can put a file like this in your Git project - **BAM!**, everyone's always in sync.

You can find a semi-exhaustive list of options in this helpful gist[152].

That's it. Time to play!

By this point, you have a working environment in which to write your code, and you understand how it does what it does. On a high level at least. And you don't need the details until you want to get fancy anyway.

That's how environments are: a combination of cargo culting and rough understanding.

What we care about is that our ES6 code compiles into ES5 so that everyone can run it. We also have it set so that our local version hot loads new code.

In the next section, we're going to start building a visualization.

[151] code_samples/env/eslintrc

[152] https://gist.github.com/cletusw/e01a85e399ab563b1236

Appendix B - Browserify-based environment

When I first wrote this book in the spring of 2015, I came up with a build-and-run system based on Grunt and Browserify. I also suggested using Bower for client-side dependencies.

I now consider that to have been a mistake, and I think Webpack is a much better option. I also suggest using one of the numerous boilerplate projects to get started quickly.

I'm leaving the old chapter here as a curiosity, and to help those stuck in legacy systems. With a bit of tweaking, you *can* use Grunt with Webpack, and Webpack *can* support Bower as a package manager.

NPM for server-side tools

NPM is node.js's default package manager. Originally developed as a dependency management tool for node.js projects, it's since taken hold of the JavaScript world as a way to manage the toolbelt.

We'll use NPM to install the other tools we need.

You can get it by installing node.js from nodejs.org[153]. Grunt, Bower, and our development server will run in node as well.

Once you've got it, create a working directory, navigate to it, and run:

```
$ npm init .
```

This will ask you a few questions and create `package.json` file. It contains some meta data about the project, and more importantly, it has the list of dependencies. This is useful when you return to a project months or years later and can't remember how to get it running.

It's also great if you want to share the code with others.

And remember, the stub project included with the book already has all of this set up.

The development server

Production servers are beyond the scope of this book, but we do need a server running locally. You could work on a static website without one, but we're loading data into the visualization dynamically and that makes browser security models panic.

[153] http://nodejs.org

We're going to use `live-server`, which is a great static server written in JavaScript. Its biggest advantage is that the page refreshes automatically when CSS, HTML, or JavaScript files in the current directory change.

To install `live-server`, run:

```
$ npm install -g live-server
```

If all went well, you should be able to start a server by running `live-server` in the command line. It's even going to open a browser tab pointing at `http://localhost:8080` for you.

Compiling our code with Grunt

Strictly speaking, we're writing JavaScript and some CSS. We don't *really* have to compile our code, but it's easier to work with our code if we do.

Our compilation process is going to do three things:

- compile Less to CSS
- compile JSX to pure JavaScript
- concatenate source files

We have to compile Less because browsers don't support it natively. We're not going to use it for anything super fancy, but I prefer having some extra power in my stylesheets. It makes them easier to write.

You can use whatever you want for styling, even plain CSS, but the samples in this book will assume you're using Less.

Compiling JSX is far more important.

JSX is React's new file format that lets us embed HTML snippets straight in our JavaScript code. You'll often see render methods doing something like this:

<<A basic Render[154]

See, we're treating HTML - in this case, an H1BGraph component - just like a normal part of our code. I haven't decided yet if this is cleaner than other templating approaches like Mustache, but it's definitely much better than manually concatenating strings.

As you'll see later, it's also very powerful.

But browsers don't support this format, so we have to compile it into pure JavaScript. The above code ends up looking like this:

[154] code_samples/old/main.jsx

```
React.render(
    React.createElement(H1BGraph, {url: "data/h1bs.csv"}),
    document.querySelectorAll('.h1bgraph')[0]
);
```

We could avoid this compilation step by using `JSXTransform`. It can compile JSX to JavaScript in the browser, but it makes our site slower. React will also throw a warning and ask you never to use `JSXTransform` in production.

Finally, we concatenate all of our code into a single file because that makes it quicker to download. Instead of starting a gazillion requests for each and every file, the client only makes a single request.

Install Grunt

We're going to power all of this with Grunt[155], which lets us write glorified bash scripts in JavaScript. Its main benefits are a large community that's created plugins for every imaginable thing, and simple JavaScript-based configuration.

To install Grunt and the plugins we need, run:

```
$ npm install -g grunt-cli
  $ npm install –save-dev grunt
  $ npm install –save-dev grunt-browserify
  $ npm install –save-dev grunt-contrib-less
  $ npm install –save-dev grunt-contrib-watch
  $ npm install –save-dev jit-grunt
  $ npm install –save-dev reactify
```

Browserify[156] will allow us to write our code in modules that we can use with `require('foo.js')`, just like we would in node.js. It's also going to concatenate the resulting module hierarchy into a single file.

Some people have suggested using Webpack[157] instead, but I haven't tried it yet. Apparently it's the best thing since bacon because it can even `require()` images.

Reactify[158] will take care of making our JSX files work with Browserify.

Less[159] will compile Less files to CSS, `watch` will automatically run our tasks when files change, and `jit-grunt` will load Grunt plugins automatically so we don't have to deal with that.

[155] http://gruntjs.com
[156] http://browserify.org
[157] http://webpack.github.io/
[158] https://github.com/andreypopp/reactify
[159] https://github.com/gruntjs/grunt-contrib-less

Grunt Config

Now that our tools are installed, we need to configure Grunt in `Gruntfile.js`. If you're starting with the stub project, you've already got this.

We'll define three tasks:

- `less`, for compiling stylesheets
- `browserify`, for compiling JSX files
- `watch`, for making sure Grunt keeps running in the background

The basic file with no configs should look like this:

```
module.exports = function (grunt) {
    require('jit-grunt')(grunt);

    grunt.initConfig({ /* ... */ });

    grunt.registerTask('default',
                ['less', 'browserify:dev', 'watch']);
};
```

We add the three tasks inside `initConfig`:

<<Less task config[160]

This sets a couple of options for the less compiler and tells it which file we're interested in.

<<Browserify task config[161]

The `reactify` transform is going to transform JSX files into plain JavaScript. The rest just tells `browserify` what our main file is going to be and where to put the compiled result.

I'm going to explain `debowerify` when we talk about client-side package management in the next section.

<<Watch task config[162]

This tells `watch` which files it needs to watch for changes and what to do with them.

You should now be able to start compiling your code by running `grunt` in the command line. If you didn't start with the stub project, it will complain about missing files. Just create empty files with the names it complains about.

[160] code_samples/old/Gruntfile.js
[161] code_samples/old/Gruntfile.js
[162] code_samples/old/Gruntfile.js

Managing client-side dependencies with Bower

Client-side dependency management is the final piece in the puzzle.

Traditionally, this is done by dumping all of our JavaScript plugins into some sort of `vendor/` directory or by having a `plugins.js` file and manually copy-pasting code in there.

That approach works fine up until the day you want to update one of the plugins. Then you can't remember exactly which of the ten plugins with a similar name and purpose you used, or you can no longer find the Github repository.

It's even worse if the plugin's got some dependencies that also need to be updated. Then you're in for a ride.

This is where Bower comes in. Instead of worrying about any of that, you can just run:

$ bower install <something>

You could use NPM for this, but Bower can play with any source anywhere. It understands several package repositories, and it can even download code straight from Github.

To begin using Bower, install it and init the project:

```
$ npm install -g bower
  $ bower init
```

This will create a `bower.json` file with some basic configuration.

When that's done, install the four dependencies we need:

```
$ bower install -S d3
  $ bower install -S react
  $ bower install -S bootstrap
  $ bower install -S lodash
```

We're going to rely heavily on d3 and React. Bootstrap is there to give us some basic styling, and lodash will make it easier to play around with the data.

All of these were installed in the `bower_components/` directory.

This is awesome, but it creates a small problem. If you want to use Browserify to include d3, you have to write something like `require('../bower_components/d3/d3.js');`, which not only looks ugly but also means you have to understand the internal structure of every package.

We can solve this with `debowerify`, which knows how to translate `require()` statements into their full path within `bower_components/`.

You should install it with:

```
$ npm install –save-dev debowerify
```

We already configured Debowerify in the Grunt config section under Browserify. Now we'll be able to include d3.js with just `require('d3');`. Much better.

Final check

Congratulations! You should now have a sane work environment.

Running `grunt` will compile your code and keep it compiling. Running `live-server` will start a static file server that auto-updates every time some code changes.

Check that your work directory has at least these files:

- package.json
- Gruntfile.js
- bower.json
- node_modules/
- bower_components/
- src/

I'd suggest adding a `.gitignore` as well. Something like this:

<<.gitignore[163]

And you might want to set up your text editor to understand JSX files. I'm using Emacs and `web-mode` is perfect for this type of work.

If `grunt` complains about missing files, that's normal. We're going to create them in the next section. But if it's bugging you too much, just create them as empty files.

You can also refer to the stub project included with the book if something went wrong. If that doesn't help, Google is your friend. You can also poke me on Twitter (@Swizec) or send me an email at swizec@swizec.com.

[163] code_samples/old/.gitignore